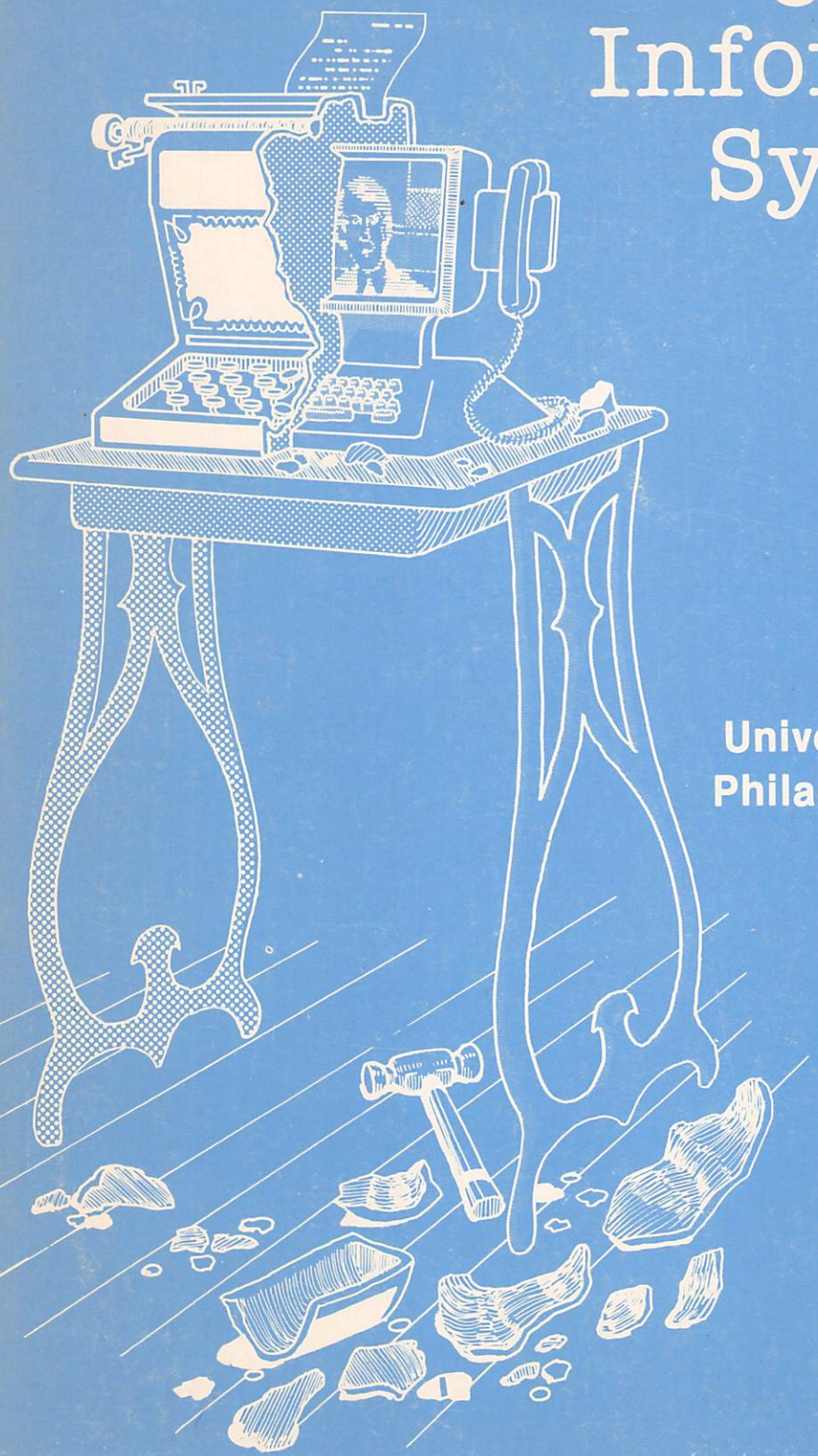
Proceedings

SIGOA

Conference On Office Information Systems



acm

University of Pennsylvania
Philadelphia, Pennsylvania
June 21-23, 1982

Edited by
J.O.Limb
ACM Order No. 611820

SIGOA NEWSLETTER
VOL. 3 • NOS. 1 and 2

Special Interest Group on Office Automation

SIGOA *Newsletter* is an informal bimonthly publication of the Special Interest Group on Office Automation (SIGOA) of the Association for Computing Machinery.

Membership in SIGOA is open to ACM members for $7.50 per year, $3 for student members, and non-ACM members for $21.50 per year. All SIGOA members receive SIGOA *Newsletter*, are given discounts at SIGOA-sponsored meetings, and may vote in the Group's elections. ACM members of SIGOA may serve as officers of the group.

Institutional or library subscriptions to SIGOA *Newsletter* are available for $17.50 per year, and the regular back issues of the *Newsletter* (when there are some) may be purchased from ACM Headquarters. SIGOA symposium and conference proceedings issues of the *Newsletter* are available, at different prices, from ACM Headquarters.

Contributions to SIGOA *Newsletter* may be sent to the Editor. They must be in machine-readable form, preferably available in someone's file system. Announcements of relevant company reports and academic theses may also be suitable for inclusion. Those intested in editing a special issue devoted to a specific topic are invited to contact the Editor of SIGOA *Newsletter* to discuss the matter. Letters to the Editor of SIGOA *Newsletter* will be considered as submitted for publication unless they contain a request to the contrary.

Technical papers appearing in this issue are unrefereed working papers, and all contributions are ordinarily to be construed as personal rather than organizational statements. Authors planning to submit to a journal any manuscript appearing in SIGOA *Newsletter* should make certain that the journal version differs sufficiently in content to satisfy the editor of that journal. All questions regarding journal policy on possible duplicate publications should be directed to the editor of the journal in question.

Changes of address and questions pertaining to the mailing of SIGOA *Newsletter* should be sent to ACM Headquarters:

ACM SIGOA
11 West 42nd St.
New York, New York 10036
212-869-7440

Executive Committee:

Chairman:
Michael P. Barnett
Brooklyn College, Dept. of Comp. & Info. Sci.
Brooklyn, NY 11210

Vice Chairman
Howard Morgan
The Wharton School of Business Administration
Department of Decision Sciences
University of Pennsylvania
Philadelphia, PA 10104

Treasurer:
Thomas F. Riesing
Ass't VP
Citibank NA
1 City Corp. Ctr.
153 E. 53rd St.
New York, NY 10043

Editor:
Michael P. Barnett
Brooklyn College, Dept. of Comp. & Info. Sci.
Brooklyn, NY 11210

SIGOA

Conference On Office Information Systems

University of Pennsylvania

Philadelphia, Pennsylvania

June 21-23, 1982

Edited by

J.O.Limb

ACM Order No. 611820

The Association for Computing Machinery, Inc.
11 West 42nd Street
New York, New York 10036

Copyright © 1982 by the Association for Computing Machinery, Inc. Copying without fee is permitted provided that the copies are not made or distributed for direct commercial advantage and credit to the source is given. Abstracting with credit is permitted. For other copying of articles that carry a code at the bottom of the first page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, P. O. Box 765, Schenectady, N. Y. 12301. For permission to republish write to: Director of Publications, Association for Computing Machinery. To copy otherwise, or republish, requires a fee and/or specific permission.

ISBN 0-89791-075-3

Additional copies may be ordered prepaid from:

ACM Order Department
P. O. Box 64145
Baltimore, MD 21264

Price:
Members: $11.00
All others: $15.00

ACM Order Number: 611820

Foreword

"What, *another* conference?", you might well ask. This question is perhaps best answered by giving a brief history of recent "gatherings" in the field. While there had been previous smaller meetings,* a workshop entitled "Office Semantics" was organized by Carl Hewitt and sponsored by MIT Laboratory for Computer Science in June 1980. Attendance was by invitation and limited to 30. The "workshop" nature of the conference led to many interesting late night discussions. An IEEE sponsored Lake Arrowhead workshop, held in September of the same year, was similar in flavor with approximately 100 invited attendees. These workshops addressed many of the needs of researchers in the field, however, the range of topics considered was limited.

Meanwhile, AFIPS (American Federation of Information Processing Societies) perceiving a need for a show and convention similar to the annual NCC (National Computer Conference) started OAC (Office Automation Conference) in 1980. This first conference attracted many outstanding professionals in the field; but, the program was primarily aimed at the needs of users and managers in areas such as word processing and data processing. OAC'81 had an "advanced technology" track in an attempt to present some of the forward looking research in the area. While these sessions were well attended the environment was not conducive to informal exchange. In summary, it seems that the workshops were too small and OAC was too large. Neither format provided a stimulating environment for researchers to present their work in the area of office automation.

There, I've used those words. No self-respecting researcher really works in the area of *office automation.* He or she probably works on database architecture, distributed computing, human interfaces or artificial intelligence. Some brave souls actually admit that they investigate "office semantics", but the semantics of that appellation still eludes me. Is "Office Information Systems" more accurate; can the work being presented at the conference be adequately described by "information systems"? Well, let's stop being pedantic, Office Information Systems is the current euphemism for you know what and that's fine. Besides, while it may not have the flair of the French "Burotique" it sounds a lot better than the anglicized version of the word which, unfortunately, rhymes with "neurotics".

Returning to history, the seeds of the present conference were sown at an ad hoc meeting held one evening at OAC'81. As I look back over my notes I see that the group informally agreed on a number of issues that set the flavor and character of the conference: I) It should be research oriented, II) It should be affiliated with SIGOA and perhaps other groups, III) It should be interdisciplinary in flavor, IV) There should be no parallel sessions (at least at the first conference), V) Attendance should be open to all through wide publicity (consistent with physical facilities), VI) It should emphasize technical quality with refereed papers appearing in a conference proceedings, and VII) It should provide maximum opportunity for informal discussion and exchange. The extent to which these goals have been met will probably be obvious by the time you read the proceedings.

Turning to the technical program, 58 papers were received in response to the open call, and each was reviewed by at least three of the program committee members. On the basis of the reviews the committee selected 21 papers for presentation. There were essentially two criteria used by the committee in making their decisions - the technical quality of the paper and the

---

* (e.g., NYU Symposium on Automated Office Systems, May 1979)

relevance to office information systems. No attempt was made to "balance" the contributions in any way. Consequently, a number of topics received little, or no coverage. The topics for the three panel sessions on evaluation of offices, office data systems and communications were chosen to fill the more glaring of these gaps. Six conference papers were selected from the program. After further review, they will be published in the new ACM Transactions on Office Information Systems, TOOIS (I think it's easiest to pronounce it twos).* An extended abstract of these papers is included in these Proceedings while a supplement containing the complete papers is being distributed to conference attendees.

We anticipate that this conference will continue annually or biennially, perhaps alternating with a European conference.** The steering committee for the next COIS is Clarence Ellis, Carl Hewett and Dennis Tsichritzis. If you would like to see this conference continue I would encourage you to contact the committee and volunteer for some of the many tasks associated with running such a conference.

I have found it very rewarding working with the organizing and program committees and I appreciate their dedicated efforts in reviewing the many manuscripts. I would especially like to thank Marylu Buono, whose efforts as Conference Secretary went well beyond the call of duty and resulted in whatever semblance of order, calm and timeliness that we may have managed to achieve.

John Limb, Chairperson

---

* The first issue of TOOIS is slated for January 1983.

** Najah Naffah, under the auspices of INRIA organized two excellent small workshops in November 1979 and October 1981. Will they be precursors to the equivalent of COIS in Europe?

ORGANIZING COMMITTEE

General Chairman	John Limb	Bell Laboratories
Vice-Chairman	Fred Lochovsky	University of Toronto
Secretary	Marylu Buono	Bell Laboratories
Treasurer	Narain Gehani	Bell Laboratories
Local Arrangements	Howard Morgan	University of Pennsylvania
	Francoise Aubert-Santelli	University of Pennsylvania
	Susan O'Leary	University of Pennsylvania

PROGRAM COMMITTEE

James Bair	Bell Northern Research	Reinhard Kofer	Siemens, Germany
Jerry Barber	MIT	John Limb	Bell Laboratories
Carolyn Cook	Xerox PARC	Fred Lochovsky	University of Toronto
Peter deJong	MIT	Najah Naffah	INRIA, France
Richard Fikes	Xerox PARC	Gary Nutt	NBI, Inc.
Irene Greif	MIT	Jeff Rulifson	Rolm Corporation
Carl Hewitt	MIT	Bing Yao	University of Maryland

STEERING COMMITTEE

Michael Hammer	MIT
Carl Hewitt	MIT
Clarence Ellis	Xerox PARC
Dennis Tsichritzis	University of Toronto

CONFERENCE ON OFFICE INFORMATION SYSTEMS
TABLE OF CONTENTS

Foreword iii

Organizing Committee v

SESSION 1 - OFFICE SYSTEMS

Chairperson: S. B. Yao

Traits: An Approach to Multiple-Inheritance Subclassing
by G. Curry, L. Baer, D. Lipkie and B. Lee 1

The Design of Star's Records Processing: Data Processing for Non-Computer Professionals (Abstract Only)
by R. Purvy, J. Farrell and P. Klose 10

Office Information Systems Research and Development Projects in Europe (Invited Paper, Abstract Only) by N. Naffah 11

SESSION 2 - MODELS

Chairperson: R. E. Fikes

Dialogue and Process Design for Interactive Information Systems Using Taxis
by J. Barron 12

Office Information Models and the Representation of 'Office Objects'
by S. J. Gibbs 21

An OIS Model for Internal Control Evaluation (Abstract Only)
by A. D. Bailey, Jr., J. Gerlach, R. P. McAfee and A. B. Whinston 27

SESSION 3 - USER INTERFACE

Chairperson: J. H. Bair

How Do People Really Use Text Editors?
by J. Whiteside, N. Archer, D. Wixon and M. Good 29

Virtual Editing: II. The User Interface
by H. A. Wilder and N. F. Maxemchuk 41

How Do People Organize Their Desks? Implications for the Design of Office Information Systems (Abstract Only)
by T. W. Malone 47

SESSION 4 - PANEL: EVALUATION OF OFFICES

Chairperson: W. Harless 50

SESSION 5 - ARTIFICIAL INTELLIGENCE

Chairperson: C. Hewitt

Extending the Power of Programming by Examples
by G. Attardi and M. Simi 52

An Actor-Based Programming System
by R. J. Byrd, S. E. Smith and S. P. de Jong 67

Supporting Organizational Problem Solving with a Workstation (Abstract Only)
by G. R. Barber 79

SESSION 6 - USER NEEDS

Chairperson: C. Cook

Toward a New Framework for Office Support
by R. R. Panko and R. H. Sprague, Jr., ... 82

Implementing Office Automation: Principles and an Electronic Mail Example
by J. H. Tucker 93

A Case Study of Office Workstation Use
by C. V. Bullen, J. L. Bennett and E. D. Carlson 101

SESSION 7 - VOICE AND TEXT PROCESSING

Chairperson: F. H. Lochovsky

Integrating Diverse Knowledge Sources in Text Recognition
(Abstract Only) by S. N. Srihari, J. J. Hull and R. Choudhari 108

Message Files (Abstract Only)
by D. Tsichritzis and S. Christodoulakis 110

A Voice Response System for an Office Information System
by D. L. Lee 113

SESSION 8 - PANEL: VIEWS OF OFFICE DATA

Chairperson: S. P. de Jong 122

SESSION 9 - FORMS

Chairperson: J. F. Rulifson

SEDL - A Language for Specifying Integrity Constraints on Office Forms
by J. C. Ferrans 123

OFFICETALK-D: An Experimental Office Information System
by C. A. Ellis and M. Bernal 131

A Modelling Tool for Office Information Systems
by J. Lebensold, T. Radhakrishnan and W. M. Jaworski 141

SESSION 10 - COMMUNICATIONS

Chairperson: K. Kümmerele

Development of a CBMS Message Transfer Protocol
by G. F. Mulvenna 153

PANEL: HOW MUCH BANDWIDTH IS ENOUGH? 160

Author Index 161

Traits :
An Approach to Multiple-Inheritance Subclassing

Gael Curry, Larry Baer, Daniel Lipkie, Bruce Lee

Xerox Corporation, El Segundo, California

Abstract : This paper describes a new technique for organizing software which has been used successfully by the Xerox Star 8010 workstation. The workstation (WS) software is written in an "object-oriented" style : it can be viewed as a system of inter-communicating objects of different *object types*. Most of the WS software considers object types to be constructed by assembling more primitive abstractions called *traits*. A trait is a characteristic of an object, and is expressed as a set of operations which may be applied to objects carrying that trait. The *traits model of subclassing* generalizes the SIMULA-67 model by permitting multiple inheritance paths. This paper describes the relationship of WS software to the traits model and then describes the model itself.

Star Workstation Software and the Traits Model

History : Star WS software has been committed to an object-oriented coding style (discussed shortly) in the Mesa programming language[Mitchell 79] since actual development first started in the spring of 1978 [Harslem 82]. Initial designs did not rely on subclassing. This was partly because the designers had had little experience with it (authors included), and partly because an extensible design based on subclassing seemed to necessitate a violation of Mesa's type system. An early Star text editor was built without the benefit of subclassing. It gradually became clear that significant code-sharing was possible if the design were based on subclassing, since the objects we were dealing with were more similar than different.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0001 $00.75

By late 1978, we had re-implemented that editor in terms of SIMULA-67-style subclassing, where object types were considered to form a *tree* under the specialization relation. The subclassing was represented as coding conventions in Mesa. That was a great help, particularly the analogue of SIMULA-67 VIRTUAL procedures (which permitted operations to be specified at more abstract levels and interpreted at more concrete ones). Use of this subclassing style extended into other areas of WS software, especially support for property sheets and open icon windows [Smith 82], [Seybold 81]; Star graphics and tables were initially designed in these terms also. As the class hierarchy grew, we began to notice that the constraint of *pure-tree* class hierarchies was causing code to become contorted, and that generalizing the concept of "class hierarchy" to include *directed acyclic graphs* would allow code to be organized more cleanly.

A new subclassing model was defined along those lines. It postulated that object types were constructed from more primitive abstractions, *traits*, corresponding roughly to SIMULA-67 classes. The major difference was that a given trait may be defined in terms of *several* more primitive ones, rather than just a single one. Supporting software - the "Traits Mechanism" - was implemented in late 1979. Star graphics [Lipkie 82] was the first major piece of Star software *designed* in terms of traits and using the full generality of the model. Other areas, especially Star folders and record files, began using the generality permitted by traits heavily.

The Traits Mechanism : A major design goal was to make the new mechanism as efficient as the old coding pattern for the case of static, tree-structured class hierarchies. We found a way to do this with a particular global optimization (outside the scope of this paper), but it required that a central facility, or *trait manager*, collect information about all extant traits. This trait manager collects information from each trait in the system regarding its storage requirements, arranges that trait's storage (in objects)

for optimum access, and mediates access to it upon the individual trait's demand. Client code (code calling the trait mechanism) adopts a Mesa coding pattern to use trait-style subclassing.

Another important property of the traits mechanism is that the cost of accessing a trait's data in an object, or the implementation of an operation that the trait introduces, is not a function of the position of the trait in the class hierarchy. There is no run-time searching.

Star today : Star software has been using the Traits Model of subclassing since 1979 with good results. Star-1 was completed in October, 1981. It defined 169 traits. Of those, 129 were *object types*, or *class traits*; i.e., 40 were purely internal abstractions. In general, each trait requires some storage in objects which carry it; 99 were of this sort. Also, each trait introduces some number of operations which can be applied to objects which carry it. While not all of these operations may be "VIRTUAL", 31 traits in Star-1 introduce this kind of operation.

The Traits Model

Object Orientation : Object-orientation is a method for organizing software where, at any time, computation is performed under the aegis of a particular object. Part of the computation may include transferring control and information to another object (message-passing), which then continues the computation; control and other information may be returned to the first subsequently. An object's state is typically represented as some sort of storage; each object has a name. A restricted form of message-passing is typically represented by procedure call, where a distinguished parameter of each procedure is the name of the object which is to continue the computation. Objects' state may be represented as records, pointers to records, names of records, implicit records, or in any number of other ways.

Subclassing : SIMULA-67 noted that often an object is a *specialization* of another, being able to to the job of the first - and more. It provided a means of expressing the common portion once, in order that the specialized object need only specify the way in which it was different from the simpler one. The specialized object *inherited* the properties of the simpler one.

The Traits model notes that an object (type) may be a synthesis of several component abstractions, being able to do the job of its components and more. It provides a means of expressing the common, or shared, parts once.

In both cases advantages come from sharing : clarity of code through factoring or abstraction; uniformity of behavior, including correctness; ease of maintenance; reduced swapping. Another important property for large systems, which both models possess, is extensibility: the addition of a new class or trait does not invalidate existing code.

Instances : There is a wide range of interpretations for the term "object". In order to avoid problems of language, we will use the term *instance* to refer to any of the objects in our universe of discourse. This is left intentionally vague.

Instances have *state*, which allows them to remember information. They also have *names*, or *handles*. Often an instance will remember the names of other useful instances.

Operations : An *operation* is a means of presenting information to and/or extracting information from an instance. Every instance possesses an identifiable set of operations, called its *operation set*. An operation is *applied* to an instance, perhaps presenting some information to the instance (in a well-defined format) and perhaps receiving some information from it (also in a well-defined format) in return. Applying an operation to an instance changes the state of the instance, in general.

Each operation has a *specification* and a *realization*; the realization *meets* the specification. The range of specifications in actual practice extends from strictly functional input/output specifications, to those including some behavioral clauses (operational specifications), to those including contextual clauses (behavior varies with context), to that which is simply "it works when you plug it in". Two operations are *equal* if they have the same specification and realization. They are *equivalent* if they have the same specification; one operation is a *variant* of another if they are equivalent.

Types : Many times instances will have the same operation set, being different only in their internal state and in their identity. The universe of instances can be partitioned into equivalence classes, based on having the same operation set (that is, two instances are in some sense equivalent if they have the same operation set). These equivalence classes are *types*. The *operation set of a type* is also well-defined.

This view says that two instances have different type if their operation sets are different, however minor the difference. While that is correct, it also ignores a lot of information about exactly *how* those operation sets are different.

Type Structure : There are many ways in which the operation sets for types can be related to those of other types.

- UNRELATED - The operation sets for all types can be totally different, so that we see no interesting type structure. This situation is supported well by programming conventions which devote one "module" to each type, and implement operations for the type's operation set within that module.

- VARIATION - It may be that there is a one-to-one correspondence between operations of one type and those of another, and that each operation is equivalent to its corresponding one. Then, each type is a *variant* of the other. This situation is supported well by the programming style which accords each object one procedure variable for each operation; realizations for each operation are recorded in the corresponding procedure variable. Streams are sometimes implemented this way.

- EXTENSION - It may be that all of the operations of one type are equal to operations of another type, but that the latter type has extra operations. Then, the latter type is an *extension* of the former. This situation is supported well by simple inheritance mechanisms.

- SPECIALIZATION - It may be that one type's operation set can be gotten from another 's by variation, perhaps followed by extension. Then, the former type is a *specialization* of the latter. This situation is supported well by SIMULA-67 and Smalltalk-80.

In the cases above, it was possible to see how the operation sets could be derived from the operation sets of other types. In the cases below, type structure is derived from units which are more basic than other types.

- UNIONS - It may be that of three types A, B and C :
  A has operations $O_1 \cup O_2$
  B has operations $O_2 \cup O_3$
  C has operations $O_1 \cup O_3$,

  where O_1, O_2 and O_3 are sets of operations. This is a somewhat contrived case. The same sort of thing happens naturally on a larger scale (indeed, perhaps *only* on a larger scale). Being minimal, the example shows more clearly what is going on. In this case, no type is a specialization of another, yet there is clearly an interesting type structure.

  Notice that the operation sets are not naturally derivable from the operation sets of other *types*, but rather from lower-level *operation subsets*. These operation subsets represent a characterization of some aspect of an instance's behavior in terms of a set of operations. The pattern arises whenever an instance has *several independent aspects*.

  A *trait* is a characterization of an aspect of an instance's behavior. The primary representation of the trait may be an natural language description of that aspect or may be some individual's intent for or understanding of that aspect. The characterization is represented by a set of specifications for operations which, considered together, embody that aspect. A set of operations with those specifications is called an *operation set* for the trait.

- SYNTHESIS - It may be that one type's operation set can be gotten from operation sets of several other *component traits* by variation of the operations in the trait operation sets, followed by union of the results, perhaps followed by extension. Then, the type is a *synthesis* of the component traits.

  This is the basic operation adopted by the Traits approach.

- RESTRICTED SYNTHESIS - It may be that one type's operation set can be gotten from those of several other *component traits* by synthesis, followed by discarding some of the resulting operations. This is not well handled by the current Traits mechanism; it can be simulated by re-defining an undesired component trait's operation to have nil realization (Note that the operation may then not meet its specification).

The discussion above has been *analytical*. It assumed instances, operations and types already existed; it tried to dissect the situation. The ensuing discussion is *constructive*. It tries to develop a view of traits as basic design units, in order to show how to incrementally build a system of trait-based instances.

Traits : A *trait* is a characterization of an aspect of an instance's behavior. It is expressed as a set of operations. Some examples of traits are:

- **IS-FORWARD-LINKED-LIST-ELEMENT** - This represents the notion that an instance carrying this trait will be linked with other instances in some forward-linked list. It specifies operations

  GetLink :
  PROC [instance : Instance]
  RETURNS [Instance], and
  SetLink :
  PROC [instance : Instance ,
  instanceLink : Instance],

  whose semantics are obvious.

- **IS-TREE-ELEMENT** - Represents the notion that an instance carrying this trait will be embedded in a tree of instances. It specifies operations

  GetParent :
  PROC [instance : Instance]
  RETURNS [Instance],
  SetParent :
  PROC [instance : Instance ,
  instanceParent : Instance],
  GetNextSibling :
  PROC [instance : Instance]
  RETURNS [Instance],
  SetNextSibling :
  PROC [instance : Instance ,
  instanceNextSibling : Instance],
  GetEldestChild :
  PROC [instance : Instance]
  RETURNS [Instance],
  SetEldestChild :
  PROC [instance : Instance ,
  instanceEldestChild : Instance],

  whose semantics are also obvious.

- **IS-NAMED** - An instance carrying this trait has a textual name. It specifies operations

  GetName :
  PROC [instance : Instance]
  RETURNS [Name], and
  SetName :
  PROC [instance : Instance , name : Name],

  whose semantics are obvious.

- **IS-PRINTABLE** - This represents the notion that the instance can print itself. It specifies the operation

  Print : PROC [instance : Instance , printer : Printer],

  which causes the instance to emit an image level representation of itself to a printer.

Note that the trait does not include realizations for the various specifications.

Simple Traits : The traits listed above are *simple traits.* A simple trait is completely defined by specifying the operations which characterize it. Figure 1 depicts a simple trait graphically.

T: IS-TREE-ELEMENT

Operation Name	*Specification*
GetParent	$S_{GetParent}$
SetParent	$S_{SetParent}$
GetNextSibling	$S_{GetNextSibling}$
SetNextSibling	$S_{SetNextSibling}$
GetEldestChild	$S_{GetEldestChild}$
SetEldestChild	$S_{SetEldestChild}$

Figure 1. Definition of Simple Trait T

Compound Traits : Sometimes a trait will be best expressed as the "sum" of other traits. For example, the trait

IS-IN-NAME-HIERARCHY =
IS-TREE-ELEMENT ∪ IS-NAMED.

specifies operations for an element of a named instance hierarchy. The operations specified by **IS-IN-NAME-HIERARCHY** might be the union of the operations specified by **IS-TREE-ELEMENT** and **IS-NAMED** individually. An instance having that trait would know that it was part of an instance hierarchy, and would know it was named. It may or may not know the same for its subordinates in that hierarchy.

In any case, it might be meaningful to augment that trait's operation set with something like

Search:
PROCEDURE [instance : Instance, name : Name]
RETURNS [Instance],

which would return the name of the subordinate having the indicated name, if there was one, and instanceNil otherwise.

We could define a new trait, **IS-SEARCHABLE**, which specifies the Search operation as its sole operation - in order to define the compound trait

IS-TREE-ELEMENT ∪ IS-NAMED ∪ IS-SEARCHABLE,

but it seems more straightforward to associate it directly with the compound trait , as in

IS-IN-NAME-HIERARCHY =
IS-TREE-ELEMENT ∪ IS-NAMED ∪ { Search }.

The latter demonstrates the *compounding method* for trait definitions. Figure 2 illustrates the compounding graphically.

The "Carries" Relation : A trait *directly carries* another trait if it is defined in terms of that trait. So, for example, **IS-IN-NAME-HIERARCHY** carries **IS-NAMED** directly. "*carries*" is the reflexive transitive closure of "directly carries", and we assume it is acyclic.

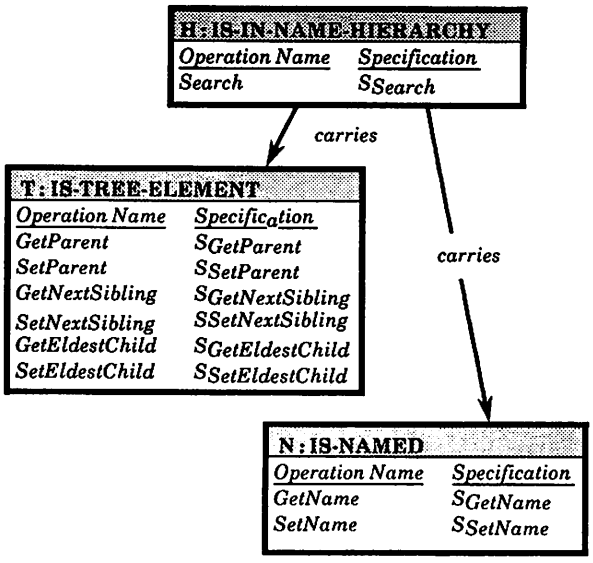


Figure 2. Definition of Compound Trait H

The Traits Graph : The collection of all traits used in a system of instances are inter-related, and form a directed acyclic graph under the "carries" relation. Nodes in the graph represent traits. Arcs represent the "carries" relation. Associated with each node in the traits graph are the specifications for operations *introduced* at that level. For simple traits, that means all of its operations. For compound traits, that means operations over and above those of the component traits. Figure 3 shows a possible trait graph.

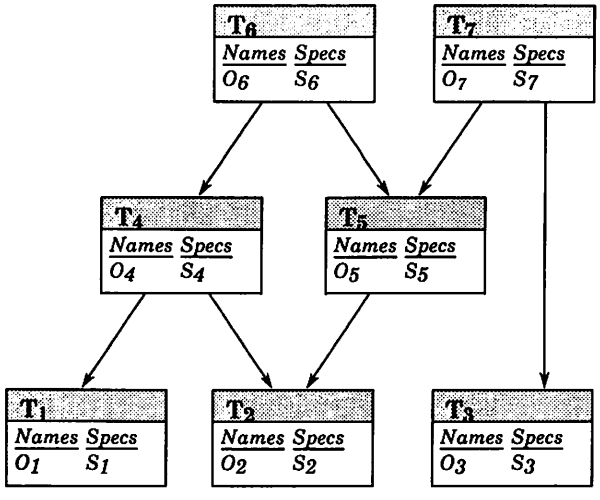


Figure 3. A Possible Traits Graph

Realizations for Trait Operations : Every trait determines a set of "carried" traits (i.e., those that it dominates in the traits graph). A trait may recommend or provide optional realizations for each of its operations - including operations introduced by the traits it carries. Figure 4 shows trait T_6 carrying traits T_1, T_2, T_4, T_5, and T_6 (itself) by displaying them in bold. The boxes adjoining each of those traits

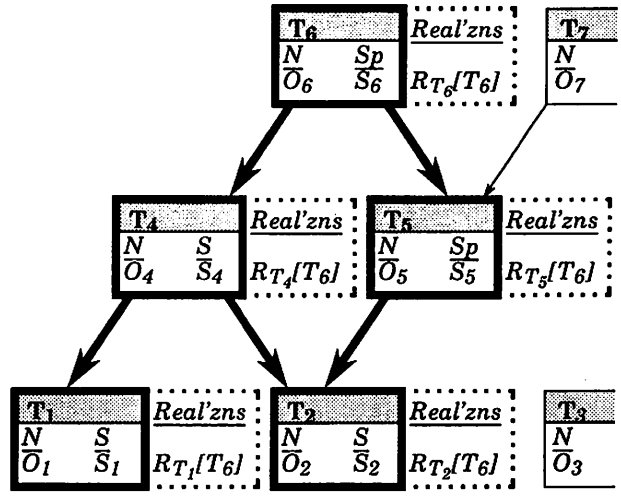


Figure 4. Realizations for Operations of Carried Traits

represent trait T_6's choices of realizations for the operations of its carried traits. The notation $R_{T_i}[T_j]$ means T_j's choices for the realizations for the operations introduced by trait T_i.

Default Realizations : A trait always assigns a *default realization* to each of the operations it introduces. The default realization may be the nil realization.

Optional Realizations : A trait sometimes makes *optional realizations* for its operations available. For any of a trait's operations, a trait may designate a pool of realizations from which other traits may choose their default realizations. This helps to maximize sharing. The default realization for an operation should be viewed as a distinguished member of the set of optional realizations for that operation. Figure 5 shows a closeup of the realizations for a particular operation o of a carried trait. The notation $r_o[T]$

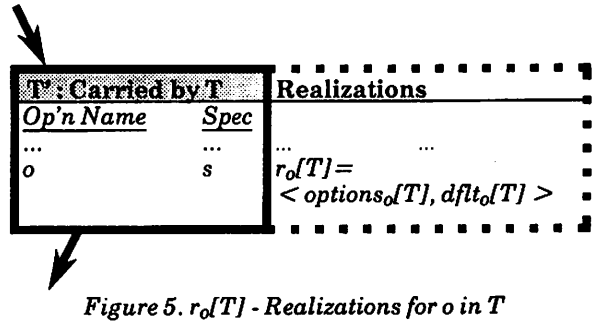


Figure 5. $r_o[T]$ - Realizations for o in T

denotes T's choices for realizations of o. $r_o[T]$ is a 2-tuple. The first element is a set of optional realizations for o from the trait T's point of view; they must all meet the specification s. The second element is a singleton or empty set of realizations from $options_o[T]$ which are T's choices for what it considers to be the default realization for o. All of the operations in $r_o[T']$ must in some sense be *defined below the level of T*.

Inheritance of Realizations for Operations : In principle, each trait in the trait graph for a system is solely responsible for determining the realizations for the operations of all of the traits it carries. In practice we find that most of a trait's choices for operations of a carried trait are exactly the choices of the traits that it directly carries for those operations. For this reason, realizations for a trait's operations are defined initially by *inheritance*.

Pure Inheritance : That is, unless the trait declares otherwise, its assignment of realizations to the operations of carried traits will be the union of assignments made by the traits it immediately carries. If those choices do not suit the trait, it must be able to *override* those assignments. The trait always has opportunity to define optional realizations for the operations that it itself introduces; it has the responsibility for defining default realizations for those operations if it can.

Suppose T is a trait in some trait graph, and that it carries a trait S which introduces an operation o. Suppose that S is carried by immediate sub-traits T_i ... T_k of T. Then we have :

$r_o[T_j] =$
$< options_o[T_j], default_o[T_j] >, for\ j = i, ..., k.$

The trait T initially views its realizations for the operation o as consisting of the union of the realizations as seen by each of the immediate sub-traits, and is potentially confused about the default realization:

$inherited\text{-}r_o[T] =$
$<inherited\text{-}options_o[T], inherited\text{-}default_o[T]>,$
where
$inherited\text{-}options_o[T] =$
$options_o[T_i] \cup ... \cup options_o[T_k], and$
$inherited\text{-}default_o[T] =$
$default_o[T_i] \cup ... \cup default_o[T_k].$

The difficulty is clear - traits T_j and $T_{j'}$ can specify different default realizations for an operation of a shared sub-trait, so that *pure inheritance* does not guarantee well-defined default realizations for operations.

Consistent Inheritance and Conflict Resolution : If o is an operation introduced by trait S carried by trait T, the realizations for o are *consistently inherited at T* iff $inherited\text{-}default_o[T]$ is a singleton or null. If this is not the case, then the trait T *must resolve the inconsistency* by explicitly designating some realization as the default. It is a *design error* for T not to do so.

Qualified Inheritance : Normally, a trait *need not* explicitly designate realizations for any but its own operations (except to resolve occurrences of inconsistent inheritance). However, it is the trait's *prerogative* to modify its realizations for any operation introduced by a trait it carries. This includes changing the default, or modifying the set of optional realizations.

Traits, Class Traits and Instances : There is a set of operation names associated with every trait T in the trait graph for a system. Those include names for operations introduced by the trait itself, as well as names for operations introduced by carried traits. Specifications exist for each of the names. Associated with each of those names is also a default realization (actually, some might be nil, but ignore that problem for now). The operation set for a trait T in a trait graph is the set of operations

$\{ o : < s, default_o[T] >,$
where o is an operation introduced by a trait carried by T, s is its specification $\}$.

It might be nice to have instances extant in the system with the same operation set as certain traits.

> In some cases, it doesn't seem to make much sense. It doesn't seem like it would be very useful to have instances whose operation set is the same as that of the simple trait IS-FORWARD-LINKED-LIST-ELEMENT; those instances would be pretty uninteresting.

Any trait having such an interesting operation set can be designated a *class trait*, and instances having the same operation set can be generated. The instance is tagged with the name of its class trait; that is the *instance's type*. The instance *carries* the same traits as its class trait.

Specifications for Trait Operations : Any operation for a trait T should have well-defined semantics. The meaning of an operation should be specified as clearly as possible when the trait is defined. That specification is the invariant part of the trait; it does not change (as do realizations for the operation) depending on which other trait is carrying T.

The specification for a trait's operation should be *in terms of the instance carrying the trait.* The client of a trait must have a clear idea of the meaning of an operation's semantics independent of its carrier.

Denoting Applications of Trait Operations : The *application of an operation to an instance* is often denoted as

<results> ← instance.operation[<parameters>].

The denotation is non-committal regarding the trait (subclass) to which the operation belongs. That has the advantage that the denotation need not change if the operation migrates from one trait to another during the course of system development. It has the disadvantage that it presents the instance as having a rather unstructured "pile" of operations, which may make the nature of the instance harder to understand. If structure in the set of operations applicable to an instance can be clearly seen, perhaps it should be expressed, as in

<results> ←
instance.operation$_{trait}$[<parameters>],

where "operation" is introduced by "trait".

Another possible form is:

<results> ←
trait.operation[instance, <parameters>].

In both of the cases above where the trait is mentioned, it is assumed that the instance carries the trait introducing the operation. The last form is well suited for use in a module-oriented language, where each trait can be represented by a single module.

Expressing Realizations for Trait Operations : It is important to find a way to express realizations of a trait's operations in a way which is independent of context (i.e., who carries that trait), so that realizations for a trait need be implemented when the trait is defined (as opposed to only when it is carried).

A realization is expressed in terms of "code" to be invoked over a particular instance I. That code may express the application of an operation of a carried trait to the same instance I. It may also involve applying operations to another instance I' of which I has knowledge (remembers, or was just told about). It may also involve changing the state of the instance somehow.

The code may also involve computations over other "objects" which happen not to be *instances* in the system in question. For example, it may involve numeric computations. While in principle "number objects" might be instances, performance considerations might recommend against it. All that is required is that the code be able to compute locally, invoke operations over instances, incorporate the results of such invocations, and change appropriate parts of the state of the instance upon which the code is operating.

Instance State vs. Trait State / Trait Data : Suppose T is a simple trait which introduces operation o with specification s, and assigns as its default realization r. Suppose i is an instance carrying T. If the specification s indicates that applying o to i will change the state of i, then it is important to ask how the realization r accesses the state it needs to change.

The problem is addressed in the Traits model by asserting that every trait carried by an instance has its own state, or storage, within the larger state of the instance itself. We go so far as to say that the state space of an instance is the product of the state spaces of the traits that it carries. Figure 6 expresses that idea graphically. Furthermore, only realizations defined

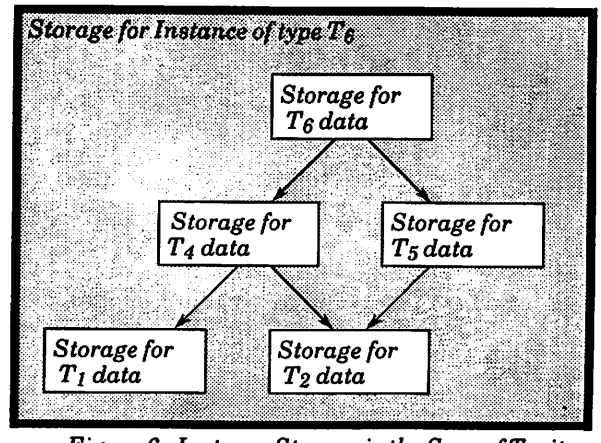

Storage for Instance of type T_6
Storage for T_6 data
Storage for T_4 data
Storage for T_5 data
Storage for T_1 data
Storage for T_2 data

Figure 6. Instance Storage is the Sum of Trait Storage

by the trait can access or modify that trait storage directly. The internal format for a trait's storage is completely up to the trait itself.

We will say nothing about the location of storage for a particular trait in instance storage. All that is important is that a realization defined to act directly on the storage for a particular trait must be able to gain access to that storage. For this purpose (and others) there is a *trait manager*, who knows how to access the storage for any particular trait, given the instance's name and the trait's name.

Instance Initialization : When an instance is generated, storage is obtained from somewhere. Embedded in that storage is storage for the individual traits carried by that instance. After the storage is allocated, individual traits are told to initialize their storage. Carried traits initialize their storage before carrying traits. In the example in Figure 6, trait T_1 would be told to initialize its storage before trait T_4 was so instructed, which would be done before trait T_6

was so instructed. The bottom-up order of trait initialization permits carrying traits to invoke carried traits operations during their own initialization.

Classes : Instances may be generated for class traits. If T is a class trait, then it needs to record its choices for realizations for all of the operations it carries. The Traits model postulates a *class* (object) for each class trait. Associated with this class is storage which records the choice of realizations. For brevity, the operation set of the class trait is called the *behavior* of the class.

Every trait which is carried by the class introduces some number of operations whose realizations can be assigned by the class. Associated with each trait T carried by a class trait T_c is enough storage to record the class trait's realizations for the operations of T. Figure 7 depicts that situation.

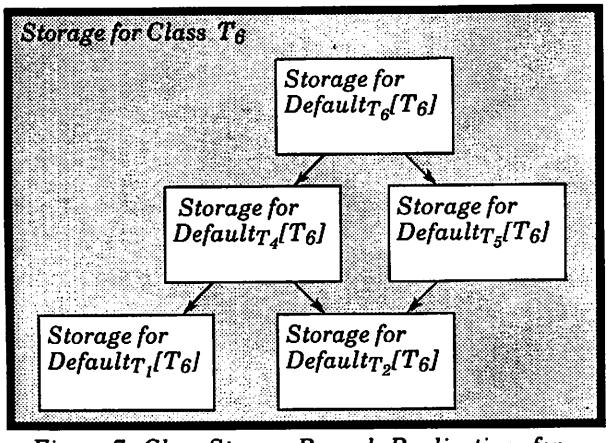


Figure 7. Class Storage Records Realizations for Trait Operations

Again, we will say nothing about the location of storage for a particular trait in class storage. All that is important is that at the time a trait operation is invoked, the realization for that operation can be found. The trait manager knows how to access a particular trait's (realizations) storage, given the name of the instance and the name of the trait.

Class Initialization : Initialization of a class is a bottom-up enumeration of that part of the traits graph dominated by the class' trait. Each trait enumerated should override any default realizations of the traits it carries and should establish its own default realizations. In order to do so, it must be able to obtain access to its component of class storage.

Instantiation : The class (object) is generally viewed as the agent which generates, or *instantiates*, instances. There may be many instances associated with a particular class, but the storage for recording the class trait's choices for default realizations is allocated only once.

Conclusions

Multiple-inheritance subclassing is a valid and useful method for organizing object-oriented software; as demonstrated by the existence of the Star Workstation. The complexity of the Star WS software has been controlled by object-orientation first, subclassing second and multiple-inheritance third.

The Traits Model is a reasonable approach to multiple-inheritance subclassing. It is possible to implement efficient supporting mechanisms, especially for statically specified class structures. The Traits mechanism is optimal for pure-tree class structures, and deep class structures cost nothing extra at run-time.

Acknowledgements : Derry Kabcenell and Tim Rentsch made useful comments during early reviews of the proposed Traits model. Eric Harslem allowed us to apply this unproven software technique to a large and important piece of software - successfully. Dan Ingalls, Alan Borning, and Dave Gifford all later noted the similarities between the traits approach and the flavors approach [Weinreb 81] of the MIT LISP machine and helped to articulate the differences. The Xerox PARC Methodology Discussion Group made plenty of interesting observations.

REFERENCES

[Harslem 82] E. Harslem and L.E. Nelson, "A Retrospective on the Development of Star," to be published in the proceedings of the *6th International Conference on Software Engineering*; Tokyo, Japan; Sept, 1982.

[Lipkie 82] Daniel Lipkie, Steven R. Evans, Robert Weissman, John K. Newlin, "Star Graphics : An Object Oriented Implementation," to be published in the proceedings of SIGGRAPH 1982.

[Mitchell 78] J.G. Mitchell, W. Maybury, and R.E. Sweet, "Mesa Language Manual," Technical report CSL-79-3, Xerox Corporation, Palo Alto Research Center, Palo Alto, California; April 1979.

[Weinreb 81] Daniel Weinreb, David Moon, LISP Machine Manual, Third Edition, March, 1981.

[Seybold 81] Seybold Report, "Xerox's Star," Volume 10, Number 16; April 27, 1981.

[Smith 82] D.C. Smith, E. Harslem, C. Irby, R. Kimball, "The Star User Interface, an Overview," to be published in the proceedings of *NCC '82*.

The Design of Star's Records Processing

Data Processing for Non-Computer Professionals

Robert Purvy, Jerry Farrell, and Paul Klose
Xerox Corporation *

Abstract

Xerox' Star Professional Workstation is distinguished by a graphic user interface committed to the What-you-see-is-what-you-get design philosophy. The system promotes a see / point / push-a-button style of interaction with immediate feedback, in marked contrast to more familiar programming or command-language interfaces.

Star's Records Processing feature integrates traditional data processing functionality into this user model. Normal Star documents may contain fields and tables. Star record files have their structure defined by copying the field structure of a regular document. Multiple output formats are provided for a record file by giving it a document with each desired format. Such a document may format multiple records in a tabular array, or present each record's data in a new copy of the formatting document. Data is transferrred between documents and records wherever fields have matching names ("field lading"); non-matching fields may be given a fill-in rule which allows summary data to be computed. Queries are expressed in an exemplary fashion, by entering desired values and ranges of values into a copy of a record (a "filter").

Common uses of the data in a record file are encapsulated in a "view", which specifies the subset of the file's records to be considered, the order in which they are to be presented, and the document to be used to format them. An arbitrary number of views may be associated with a record file; each may have its own index, but the underlying records are stored only once.

Common records processing functions are provided at a high-level user interface: Moving a document, a folder of documents, or another record file to a record file adds records to the file. Moving a record file to a printer causes the current view of the record file to be printed.

Benefits include an economy of concepts and effort for user and implementor alike. along with the synergy of a unified environment. The approach seems to extend easily to a number of new domains. However, increased efforts are required to produce a coordinated design and implementation throughout the whole system, and acceptable performance may be harder to achieve.

OFFICE INFORMATION SYSTEMS RESEARCH AND DEVELOPMENT PROJECTS IN EUROPE
(Invited)

Najah Naffah

INRIA, France

Abstract

In Europe most research projects in the area of Office Information Systems are concentrated around public communication services and transmission networks. Very few projects exist in what we call the Automated Office of the Future.

In the first category, known as TELEMATIQUE, we find the videotex services, which consist of data banks connected to public networks and offering access to residential users through their color TV sets. A second service, called TELETEX, will start soon in different European countries to replace the present telex service with some kind of communicating word processors. All these projects are sponsored by the PTT and operate according to a set of standards elaborated by the CCITT.

On the other hand, true Office Information Systems projects, which are tailored to the private users and organizations needs, are very rare. Some manufacturers are working on new word processors combining data processing and text processing, with voice input for command, and voice output for signalling.

Local networks have been built in different research centers and have been produced by local manufacturers and distributed on national basis (e.g., Cambridge Ring in England, Danube in France, and Cobus in Switzerland). While manufacturers as ICL, Thomson, Siemens, Olivetti are announcing products with many components imported from the U.S., the research community has been very active in exploring new approaches for O.I.S.

Under the sponsorship of the CEC, a group of researchers from France, Germany, Norway, Sweden, U.K., are working on the interconnection of Heterogeneous message systems, where the most difficult problems reside in the naming strategy to adopt, the conversion between message format, the high level protocols to implement.

Two research centers in Switzerland and in France, have been building workstations for the office. The first one has produced a portable personnel computer with text editor (Smaky - Ecole Polytechnique de Lausanne), and the second has built a sophisticated workstation with many integrated services (Buroviseur of the Kayak project - INRIA, France). In addition, a complete prototype of O.I.S. is under development in this project.

Other work is undertaken by different research centers on issues related to O.I.S. (Queen Mary College - London; GMC, HMI and the University of Hamburg - Germany; INRIA, IMAG, SUPELEC, UPS - France, ...).

In this presentation, we will give a summary of the work that has been accomplished and the future of some of the projects.

Dialogue and Process Design for Interactive Information Systems Using Taxis

John Barron

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada

Abstract

This paper deals with dialogue and process management for interactive information systems (abbrev. IISs) within the framework of Taxis <Mylopoulos et al 80>, a language for IIS design. Dialogues between a user and the system are represented through a small set of primitives incorporated into Taxis while process control is accomplished by incorporating Hoare's I/O commands for communicating sequential processes <Hoare 78>. The overall organization and structure of dialogue and process control for a particular IIS is achieved using scripts, a modified version of augmented Petri nets <Zisman 77>, and the Taxis conceptual framework which stresses generalization abstraction. A journal editing procedure is used to illustrate the proposed extension.

Acknowledgements

I wish to thank Professors John Mylopoulos and Alex Borgida for their interest, guidance, and invaluable criticisms while doing the research reported in this paper. Also many thanks to Sol Greenspan for his many helpful suggestions during this paper's preparation. I would also like to acknowledge financial support from the University of Toronto Computer Science Department and the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

1.1 Motivation

We are concerned with the design of interactive information systems (abbrev. IISs). The importance of IISs lie in their widespread commercial applications, such as systems for airline and hotel reservations, inventory control, and credit card verification. This importance is further underlined by the increasing acceptance of office automation in today's businesses. We see an IIS as being comprised of a large database of facts, a collection of transactions acting on the database, a set of semantic constraints specifying all possible database states, and a user interface facility.

We view IIS design to be, in large part, the construction of a conceptual model that describes a database, the possible transactions defined on it, and the user/database interactions to be supported. We believe that describing an IIS at a conceptual level allows the use of design methodologies not possible at lower levels, thus making it easier for a designer to organize and integrate details into a coherent whole. Mapping a conceptual model to an implementable program can be done by the computer itself.

At the heart of most design methodologies lie one or more abstraction techniques. The most common abstraction technique is aggregation abstraction, which forms the core of software design methodologies such as "stepwise refinement through decomposition" (see <Wirth 71>). We are interested in a methodology for software design that uses generalization abstraction <Smith and Smith 77> and is called "stepwise refinement through specialization" (see <Borgida et al 81>). This methodology allows us to suppress the difference between a number of related concepts (eg. EDITOR, AUTHOR, REFEREE) and, instead, to consider a more general concept (eg. PERSON) which contains all the instances of its subconcepts and which factors out only the things that are common to all its subconcepts.

1.2 Overview

Previous work by <Mylopoulos et al 80> and <Wong 81> has resulted in the design of that part of Taxis that offers tools for the specification of the semantic component of an IIS, i.e. the description of the data structures and transactions constituting the heart of an IIS. This paper is primarily concerned with the design of a second part of Taxis which allows the specification of the pragmatic component of an IIS, i.e. the description of the processes that conduct dialogues with the users and interact with the database as a result of that dialogue.

The parts of a Taxis program that are used to represent these processes are called scripts. Another extension of Taxis is currently being investigated which will allow the specification of a syntactic component of an IIS, i.e. the specification of the allowable syntactic forms for messages exchanged between an IIS and its users <Pilote 82>.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0012 $00.75

The pragmatic component of Taxis serves as the control mechanism for initiating, conducting, and terminating dialogues between an IIS and its users. That is, it restricts the possible long term sequences of actions the system can perform. Which particular actions are actually performed depends on the user's input and the current state of the system. The pragmatic component also provides the context under which dialogues take place. This means that a dialogue between a user and an IIS can only proceed in expected ways. For example, the editor of a journal cannot accept a paper for publication before that paper has been refereed. Thus the pragmatic component of Taxis provides the organization and structure for user/IIS dialogues and serves as the interface between the syntactic and semantic components of Taxis.

2 Background

2.1 Taxis Concepts

It is not the purpose of this paper to describe the tools offered by Taxis for the design of the semantic component of an IIS (see <Mylopoulos et al> for this). In this section we review some of the basic concepts necessary for understanding the remainder of the paper.

The basic types of objects in Taxis are tokens, classes, and metaclasses. Tokens are the constants of a Taxis program and represent particular entities in the enterprise being modelled. A class is a collection of tokens sharing common properties. Thus EDITOR is a class representing the concept of an editor and can have in its extension tokens, such as joe-editor, representing particular editors. Metaclasses are similar to classes in every respect except that their instances are classes rather than tokens. Thus predefined metaclass VARIABLE-CLASS can have in its extension classes such as EDITOR, AUTHOR, and REFEREE that serve as database relations. Similarly, TRANSACTION-CLASS is a metaclass that has transactions, i.e. procedures, in its extension. For example, a transaction ADD-REFEREE might insert referee tokens into a list represented by class REFLIST provided certain prerequisite conditions are satisfied.

Classes and tokens have properties through which they are related to other classes and tokens. Properties are represented as triples, consisting of one or more subjects, an attribute or property name and a property value (abbrev. p-value). Factual properties represent specific facts about tokens. For example, "joe-editor.address" returns 38 EDITORS LANE for the property (joe-editor, address, 38 EDITORS LANE). Definitional properties provide structural information about classes. Thus "EDITOR..address" returns as its p-value the class ADDRESS for the property <EDITOR, address, ADDRESS>. The definitional properties of a class induce factual properties for their instances. Thus if joe-editor is an instance of EDITOR and <EDITOR, address, ADDRESS> is a definitional property of EDITOR, then joe-editor must have a corresponding factual property (joe-editor, address, x) where x is an instance of ADDRESS. Property categories are used to express what aspects of the structure and behaviour of a metaclass each of its properties convey. For example, a transaction class has parameter, local, prerequisite, action, and return properties which are used to describe the parameter and local variables of a transaction, a set of prerequisite conditions that must be true, a list of actions to be carried out, and a value to be returned upon completion of the transaction.

The Taxis exception handling mechanism is a modified version of Wasserman's procedure-oriented exception handling <Wasserman 77>. Exceptions are instances of excertion classes. When a prerequisite property in a transaction evaluates to false, an instance of a specified exception class is raised, i.e. created, and passed as the only argument to an exception handler which is either another transaction or a script. For example, if during the execution of the ADD-REFEREE transaction, an ILLEGAL-REFEREE exception is raised, then ADD-REFEREE is terminated and the newly created exception instance is passed to an exception handler script FIND-NEW-REF which examines the circumstances causing the exception and conducts a dialogue with the editor to resolve it. Further details of the Taxis exception handling mechanism can be found in <Mylopoulos et al 80>.

2.2 The Generalization Hierarchy of Classes

As we have already mentioned, generalization abstraction is particularly relevant for IIS design and facilitates a new design methodology for IISs which we call stepwise refinement through specialization <Wong 81>. We see this as being used in conjunction with stepwise refinement through decomposition <Wirth 71>. A Taxis program is viewed as a collection of classes organized into a generalization hierarchy. This rule applies for data, transaction, and exception classes.

Thus SCIENCE-EDITOR is a specialization of EDITOR and therefore every instance of SCIENCE-EDITOR is also an instance of EDITOR (abbrev. SCIENCE-EDITOR IS-A EDITOR). Clearly, every definitional property of EDITOR is also a definitional property of SCIENCE-EDITOR. Moreover, SCIENCE-EDITOR can have additional properties that EDITOR does not have at all or it can redefine some of the properties of EDITOR. For example, SCIENCE-EDITOR could inherit "name", "address", and "phone#" properties of EDITOR but redefine a "profession" property to be instances of class SCIENTIST rather than just PROFESSION. (Note that SCIENTIST IS-A PROFESSION must hold for the specialization to be valid.)

3 Designing the Pragmatic Component of an IIS

3.1 Introduction to Scripts

As mentioned before, processes involving user/system interaction are represented by scripts. A script can be thought of as a known plan to accomplish some goal. Each script is represented using a transition net. That is, a script is a linked set of states that form one or more paths, some of which may be cycles. The decision of which particular path to follow is determined by user input (i.e. dialogue acts) and the current state of the system, making scripts both event and data driven.

Because IIS applications involve potentially many users conducting a large numbers of simultaneous operations on a database, it is desirable that instances of Taxis scripts be able to run concurrently. As a result, tools are needed for modelling possible coordination and data transfer requirements between script instances. We have adopted a

modified form of Hoare's I/O commands for communicating sequential processes <Hoare 78>. These commands, which we call inter-script commands, can be used both to provide coordination and to transfer data between concurrently running script instances.

The script formalism used to model processes is based on a simplified version of Zisman's augmented Petri nets <Zisman 77>. As such, scripts provide facilities for modelling decision making, concurrency, and synchronization.

A script can be viewed as a directed graph with two types of nodes. A circular (or elliptical) node is called a state and a bar node is called a transition. Each transition can have one or more input states (states that have edges directed into the transition) and one or more output states (states that have edges directed out of the transition). Similarly, each state can have one or more input transitions (transitions that have edges directed into the state) and one or more output transitions (transitions that have edges directed out of the state). Each state of a script can be either on or off (said to be either marked or unmarked). If all the input states of a transition are marked, then the transition is said to be enabled. An enabled transition may fire. The firing of an enabled transition unmarks each of the input states and marks each of the output states. (A currently marked state that is remarked as a result of a transition firing simply remains marked.) The firing of transitions generally disables some transitions and enables others.

Decision making in a script is provided for by a state having multiple output transitions. Concurrency can be expressed by a transition having multiple output states while synchronization can be expressed either as a transition having multiple input states or a state having multiple input transitions.

Each transition in a script has a set of associated conditions and actions (*). An enabled transition is fired after the successful execution of all its actions. These actions are executed only if all its conditions have been evaluated to true beforehand. If all the conditions of an enabled transition are not currently true one of two things may happen: the conditions will become true at some later time or, as the result of other transitions firing, the transition may become disabled. The set of enabled transitions of a script are called the active transition set. The enabling and firing of transitions changes membership in this set. As such, the enabling and firing of transitions represents the occurrence of events. Completion of these events may require communication with outside sources, such as with script users or with other concurrently running scripts. These types of communications (i.e. dialogue acts and inter-script coordination with or without data transfer) are carried out as actions in the action sets associated with the transitions.

(*) Our formalism differs from <Zisman 77> in that he associates one or more condition/action pairs with each transition while we only associate a single list of conditions followed by a single list of actions with each transition.

3.2 Script Semantics

Scripts are "transition" based, in that transition descriptions make up the body of a script. In Taxis, scripts are treated as classes similar to transaction classes in that they have parameter and local property categories, but differ in that the body of a script is composed of transition properties. To illustrate the concept of scripts we present the FIND-NEW-REF script exception handler mentioned earlier. This script conducts a dialogue with the editor, informing him of the circumstances surrounding the referee's rejection and requesting a suitable replacement. Once the editor has supplied a new referee the script adds him to the referee list using the ADD-REFEREE transaction. The script is shown on diagram 1.

FIND-NEW-REF DIAGRAM 1

Values are passed to a script instance through its parameter list property. These values uniquely identify an instance of that script. Because many instances of a script can exist simultaneously, their parameter values must be unique. The parameter list for FIND-NEW-REF is given as:

```
SCRIPT-CLASS FIND-NEW-REF with
  parameters
    find-new-ref: exc-parm
```

This sets up the definitional property <ILLEGAL-REFEREE, find-new-ref, FIND-NEW-REF>. Particular instances of ILLEGAL-REFEREE define a factual property whose p-value is the current execution instance of FIND-NEW-REF.

Local properties are used to declare all the local variables of a script, including parameters. State variables, whose names by convention are the same as the states with which they correspond in the script's definition, can have values 'on' or 'off', both belonging to the class STATE. If a state variable has a value 'on' then that state is said to be marked; if it has a value 'off' then it is said to be unmarked. A state variable initially has value unknown, meaning that it is neither marked nor unmarked. When determining if a transition is enabled, states with the values 'off' and unknown are treated the same, i.e. as unmarked; however the distinction between the two values has important consequences when studying the past execution of a script. Some of the variables that might be declared in the local property category of FIND-NEW-REF are:

```
locals
  exc-parm:ILLEGAL-REFEREE
  new-ref:REFEREE
  illegal-referee initially 'on'
  editor-replied, have-new-ref:STATE
```

The "exc-parm" property serves as the parameter of the script. The "new-ref" property is a local variable (which by default initially has value unknown). The "illegal-referee", "editor-replied", and "have-new-ref" properties serve as the

states in the script. The "illegal-referee" state is initially marked (i.e. has initial value 'on') while the "editor-replied" and "have-new-ref" are initially neither marked nor unmarked (i.e. have value unknown).

Each transition property in the transition category corresponds directly with the transition of the same name in the script's definition (see diagram 1). A transition property has a transition class instance as its value. Transition classes have input, output, condition, and action property categories. The input and output properties of a transition specify the input and output states of that transition. Condition properties are Boolean expressions without side effects while action properties are non-Boolean expressions with side effects. These Boolean and non-Boolean expressions include the usual expressions allowed in Taxis (see <Mylopoulos et al 80> and <Wong 81>) as well as additional expressions related to dialogue and inter-script communication.

The FIND-NEW-REF script has two transition properties in its transition list category. The first, "tell-editor", passes the exception data to the editor and requests a new referee from him. The second, "add-new-ref", inserts that referee into REFLIST using the ADD-REFEREE transaction.

```
transitions
  tell-editor:
    input from:illegal-referee
    output to:editor-replied
    actions
      ref-rejected:
        s-inform(exc-parm.e.code,
          exc-parm.r,exc-parm.a,
          exc-parm.p)
      get-new-ref:
        s-request(exc-parm.e.code,
        new-ref)

  add-new-ref:
    input from:editor-replied
    output to:have-new-ref
    conditions
      ed-supplied-new-ref?:
        u-informed(exc-parm.e.code,
        new-ref)
    actions
      insert-in-list:
        add-referee(new-ref,exc-param.a,
        exc-parm.e,exc-parm.p)
        exc-handler eh1 for
        ILLEGAL-REFEREE is FIND-NEW-REF
end /* transition list */
```

When a script instance is first created (as a result of either an instantiate command or a raised exception), the active transition set is determined as those transitions with all their input state properties initially having value 'on'. Since "illegal-referee" has initial value 'on' and is the only input state of the "tell-editor" transition, that transition is initially the only transition in the active transition set. Each script instance has an interpreter instance associated with it which executes the script instance by checking and rechecking all the conditions of all the transitions in the active transition set. When a set of conditions belonging to an enabled transition are found to be true, the interpreter stops condition checking and executes and fires that transition. Thus the "tell-editor" transition is immediately executed upon instantiation of the FIND-NEW-REF script as it has no condition properties associated with it, i.e. the condition true is assumed. Once a transition has been executed it is fired by assigning the value 'off' to all input state variables and the value 'on' to all output state variables. This usually results in removing some transitions from the active transition set and replacing them with others. Firing the "tell-editor" transition (after its successful execution) results in the removal of that transition from FIND-NEW-REF's active transition set and the addition of the "add-new-ref" transition in its place. Condition checking is now resumed on the new active transition set. This execution and firing cycle for the transitions of a script continues until either the script instance is explicitly terminated using a terminate command or is implicitly terminated by an empty active transition set. The FIND-NEW-REF script is implicitly terminated once the "add-new-ref" transition fires as at this time its active transition set is empty. It should be noted that condition sets in the active transition set are continually checked and rechecked even if none of them are presently true (*). They may later become true as the result of some dialogue act, inter-script communication, or timing event.

A transition is carried out by executing all its action properties in sequential order. The execution of the "tell-editor" transition thus involves executing its "ref-rejected" and "get-new-ref" action properties. The "ref-rejected" action uses an s-inform command to pass the "r", "a", and "p" p-values of FIND-NEW-REF's exception instance argument (representing referee, author, and paper exception information respectively) to the editor's terminal, identified by his code. The "get-new-ref" action, on the other hand, simply requests the same editor to supply a new referee. Execution of the "add-new-ref" transition can proceed only if its "ed-supplied-new-ref?" condition is satisfied. This condition checks if the editor has indeed supplied a new referee to the script instance in response to the s-request command. The editor can supply such a value by using a u-inform command to supply values to new-ref's properies, i.e.

```
u-inform(exc-parm.find-new-ref,
  new-ref.name<--"some name",
  new-ref.address<--"some address",
  new-ref.phone<--"some phone number",...)
```

The first argument of this command uniquely identifies the FIND-NEW-REF script instance via its parameter factual p-values. The remaining arguments of the command supply actual values to the "new-ref" local property of that script instance. The "insert-in-list" action of the "add-new-ref" transition calls the ADD-REFEREE transaction with the new referee value as one of its arguments. It is interesting to note that this script specifies itself as the exception handler for the ILLEGAL-REFEREE exception class that may be raised during ADD-REFEREE's execution. This means that a potentially infinite number of FIND-NEW-REF instances may be created, one for each of a series of rejected referees.

(*) The order and rate at which enabled transitions are checked in some script is undetermined. We assume that the firing order of several simultaneously transitions is unimportant; if this is not the case then the desired order can be built into the script's definition.

Because of the cyclic nature of condition checking in a script instance, scripts can "live" for long periods of time. For example, script instances that monitor the actions of either an editor or a referee during the review of a paper for publication in some journal can realistically run for several months or longer. Because of this, we can examine the factual properties of a script in much the same way as we retrieve data from a relation. For example, we can retrieve the referee, author, and paper information for all unresolved ILLEGAL-REFEREE exceptions being handled by joe-editor (via the FIND-NEW-REF script) in the following manner:

```
for script-instance in FIND-NEW-REF
where script-instance.exc-parm.e =
    joe-editor do
  s-inform(joe-editor.code,
     script-instance.exc-parm.r,
     script-instance.exc-parm.a,
     script-instance.exc-parm.p)
```

Of course, "script-instance" must be declared as a local variable in the script using it, i.e. "script-instance" must have FIND-NEW-REF as its definitional p-value. This is so "script-instance" can have instances of FIND-NEW-REF as its factual p-values.

We can also examine the factual p-values of a script's local properties. For example, to determine the execution status of an instance of FIND-NEW-REF we can use an u-request command to examine that script's state properties:

```
u-request(exc-parm.find-new-ref,illegal-
    referee,editor-replied,have-new-ref)
```

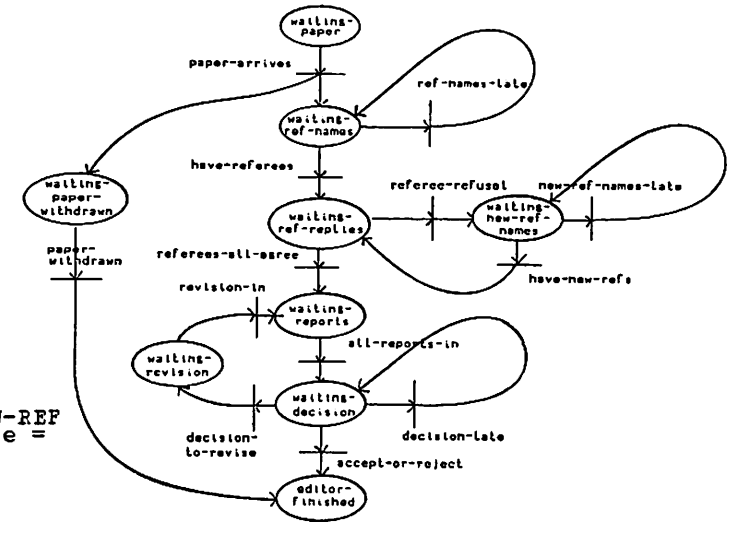
If this command returned values 'off', 'on', and unknown respectively for the three state properties "illegal-referee", "editor-replied", and "have-new-ref" then we would know that the first transition "tell-editor" has successfully fired but that the second transition "add-new-ref" has not yet fired (most likely because the editor has not yet supplied a new referee value). The factual p-value of any condition or action property of a script is always nothing, i.e. it is undefined.

4 A Journal Editing Example

4.1 The Journal Editing Procedure

The example presented here is based on a modified version of Zisman's journal editing procedure as outlined in chapter 6 of <Zisman 77>. Basically, the process of publishing a paper in a journal requires the journal's editor to appoint one or more referees to review the paper. Each referee submits a report to the editor on the suitability of the paper, and the editor decides to accept, reject, or ask for revisions. Decisions involving revision require the referees to submit new reports on the revised paper. The process concludes when the editor makes a decision to accept or reject. We use two scripts to model this procedure; one for the editor's actions and one for each of the referee's actions. Each time a paper enters into the system, an instance of the editor script and one or more instances of the referee script are used to monitor the activities of the editor and the various referees for that paper.

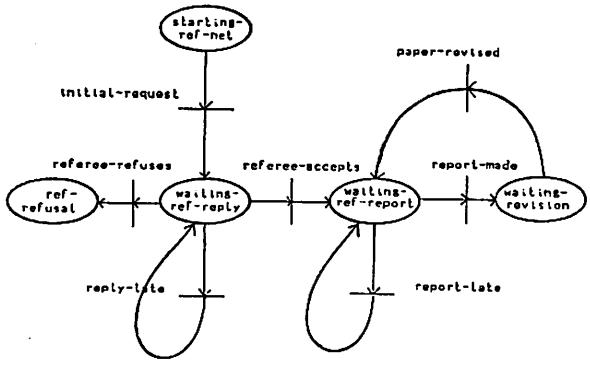
The editor script, EDITOR-NET, is shown in diagram 2. Once a paper arrives,



EDITOR-NET DIAGRAM 2

the editor must designate one or more referees. Each time a week expires and he has failed to do so, the system sends a reminder to him. Once the referees have been designated, an instance of the referee script is created for each referee. If one or more of the referees should refuse to review the paper, the editor is required to find replacements. Each time a week passes in which the editor hasn't yet designated new referees to replace any refusing referees results in the system sending a reminder to this effect to the editor. Referee script instances are created for all new referees. Once all proposed referees have agreed to review the paper, the editor waits for their reports. After the editor has received all these reports he must make his decision within two weeks; otherwise the system sends him a reminder for his overdue decision every two weeks until he has made it. If, on the one hand, the editor's decision calls for a revision of the paper, the author is informed of this and the editor awaits the revision. Once the revised paper is received, it is passed to each of the referees and their new reports are awaited. When the editor receives these reports he must again make a decision regarding the suitability of the paper for publication. If, on the other hand, the editor's decision is to accept or to reject then the author is informed of the decision and the editor and referee script instances are terminated. At any time after the editor receives a paper right up until the time the editor has made a decision of either accept or reject, the author has the option to withdraw his paper, in which case the editor and referee script instances are terminated.

The referee script, REFEREE-NET, is shown in diagram 3. Once an instance of the referee script has been created for a proposed referee, the referee is asked for his consent. If his reply is no, then this fact is communicated to the editor script and the referee script instance is terminated. Each time a two week period passes with no reply from the proposed referee the system sends a reminder to that referee. If the proposed referee's reply is yes than his report is awaited. Again, each two week period that passes



REFEREE-NET DIAGRAM 3

without the editor having received the referee's report causes the system to send that referee a reminder. Once the referee's report has been received and this fact has been communicated to the editor script, the referee script waits for the possibility that the editor's decision may be to revise (in which case a revised paper should be forthcoming). If the editor's decision is to accept or to reject then the editor script terminates all instances of the referee script as well as itself.

4.2 The Editor and Referee Scripts

The complete EDITOR-NET and REFEREE-NET scripts are given in <Barron 80>. We present only fragments here to further illustrate the various features of scripts.

Each time a paper is submitted for publication, the editor (or his secretary) creates an instance of EDITOR-NET for it. The first transition of this script, "paper-arrives", conducts a dialogue with the author and the editor; informing the author of the paper's receipt and of his option to withdraw it at any time and requesting the editor to supply one or more referees to review the paper. Consider the coding of the this transition:

```
paper-arrives:
  input  from:waiting-paper
  output to:(waiting-ref-names,
    waiting-paper-withdrawn)
  actions
    paper-recieved: paper-recieved<--true
    tell-author: s-inform(author.code,
      paper-recieved)
    paper-withdrawn?: s-request
      (author.code,paper-withdrawn)
    get-ref-names: ask(editor.code,
      not(nothing =
        (get-object rflt in REFLIST
        where((rflt.editor = editor)
          and(rflt.author = author)
          and(rflt.paper = paper)))
    all-ref-names-submitted?:
      s-request(editor,code,
        all-ref-names-in)
    set-wait-ref-time: ref-time <--
      current-time
  end;
```

The only input state, "waiting-paper" of the transition is the only state variable initialized to value 'on' in the local properties of the script. This means that initially the "paper-arrives" transition is the only enabled transition in the script. Since it does not have any condition properties (the default condition true is assumed) its action properties are executed sequentially and it is fired. Firing of this transition marks the "waiting-paper-withdrawn" and "waiting-ref-names" states and unmarks the "waiting-paper" state. The effect of this is that three transitions, "paper-withdrawn", "have-referees", and "ref-names-late", are enabled and the "paper-arrives" transition is disabled. It is interesting to note that the "paper-withdrawn" transition will remain enabled until it is either fired or the script is terminated. This allows the author to withdraw his paper anytime before the editing procedure has been completed.

The s-inform, s-request, and ask commands in the action properties are examples of a script performing dialogue acts. If we assume that a user's code is simply his name then

s-inform(joe-author,paper-received)

informs the user identified by code, joe-author, of the value of the "paper-received" property, i.e. it informs the author that his paper has been received (if the value of "paper-received" is true). Similarly

s-request(joe-author,paper-withdrawn)

requests joe-author to supply a value of paper-withdrawn (if he wishes). To respond to that request the author may execute a u-inform command. For example if joe-author desires to withdraw his paper he can execute

```
u-inform((joe-editor,joe-author,paper1).
    editor-net,paper-withdrawn<--true)
```

The first argument of any u-inform must uniquely identify the script instance; the remaining arguments must be name/value pairs. Two points should be kept in mind about s-request commands. First, the execution of any s-request simply passes the request for data to the appropriate users; it does not require that the requested data be supplied at any time in the future. Thus an author submitting a paper for publication can withdraw it from consideration at any time by simply using an u-inform to assign the value true to "paper-withdrawn". If the author never wishes to withdraw his paper he can simply do nothing or he can assign a value of false to "paper-withdrawn" using a u-inform command. If he assigns false to "paper-withdrawn" the s-request command for that value is finished, i.e. the author cannot later withdraw his paper by reassigning true to "paper-withdrawn". Second, an s-request command must be executed for any variable that is supplied a value by a u-inform command. This ensures the script's "integrity" by maintaining an orderly dialogue between a script and its users. Without this condition, a user could arbitrarily assign (and re-assign) values to a script's variables without any regard to the effect this might have on the script's execution.

A user may examine the values of any of a script's variables during its lifetime by using a u-reguest command. Its form is the same as an s-request command except that its first argument identifies a script instance rather than a script user. For example

```
u-request((joe-editor,joe-author,
    paper1)..editor-net,paper.title)
```

will return the title of the paper written by joe-author (which is represented by

token paper1) that is being considered for publication by joe-editor.

The ask command is used by a script to ask a user to perform an action. The first argument identifies a user while the remaining arguments are Boolean expressions. If a Boolean expression is true the ask command does nothing. However, if it is false, the expression is passed to the user who may make it true. There is no requirement that the user actually make this expression true. To ensure that he does, the script must make an explicit check at some later time. The ask command in the above transition checks if the REFLIST relation has any referees that have been assigned to review the author's paper. If not, the editor is asked to insert some referees in the relation for that author (as a way of making the passed expression true). No means are provided for the editor to directly do such an insertion; instead he must use some previously defined script. This script would conduct dialogue with the editor to obtain referees and use the ADD-REFEREE transaction to insert each referee in REFLIST. (As such, this script would be very similar to FIND-NEW-REF.) The use of such a script is necessary so that control can be maintained by the system as to who performs what operations on the database (and with what data). Once the editor has designated the desired number of referees he uses a u-inform command to assign value true to the "all-ref-names-in" local variable in the editor script.

It should be noted that the actual manner in which a dialogue command is performed depends on whether or not the user has direct contact with the system. Dialogue with direct users (for example, the editor) can be performed directly on terminals. Dialogue with indirect users (for example, authors and referees) might be conducted by mail or telephone with perhaps a secretary (who is a direct user) acting as the interface.

To gain information about dialogue previously conducted, we have five predicates: one for each of the five dialogue commands discussed above. For example

u-informed(joe-author,paper-withdrawn)

returns true if the author has performed a u-inform command supplying a value for "paper-withdrawn" and false otherwise. In general, a u-informed predicate checks if the user named as its first argument has executed one or more u-inform commands supplying values to the rest of the predicate's arguments. The other four predicates: u-requested, s-informed, s-requested, and asked work in much the same way as u-informed, but instead check if appropriate u-request, s-inform, s-request, and ask command(s) have been previously executed.

Another interesting transition is the "have-referees" transition. This transition checks if referees have been proposed to review the author's paper and, if so, it creates an instance of REFEREE-NET to monitor the activities of each such referee.

```
have-referees:
  input: from:waiting-ref-names
      all-refs-agree?: /* This condition is
          true only if all proposed referees
          agree to review the paper. */
          (begin
             locals agree:BOOLEAN
             actions
               a1: agree <-- true
               a2: for rflt in REFLIST
                   where((rflt.editor=editor)
                     and(rflt.author=author)
                     and(rflt.paper=paper))do
                   if giving((rflt.referee,
                     author,editor,paper).
                     referee-net,reply:YES)=
                     false
                   then begin
                     a21:agree<--false
                     a22:exit
                   end
             returns rtn: agree
          end) = true
  actions
     take-replies: for rflt in REFLIST
          where((rflt.editor = editor)
            and(rflt.author = author)
            and(rflt.paper = paper))do
          take((rflt.referee,author,editor,
            paper).referee-net,reply:YES)
  end;

referee-accepts:
  input from: waiting-ref-reply
  output to: waiting-ref-report
  conditions
     ref-accepts?: ref-reply = 'yes'
  actions
     tell-editor: s-inform(editor.code,
       referee,ref-reply)
     tell-editor-net: give((editor,author,
       paper).editor-net,ref-reply)
     set-time: report-time<--current-time
     ask-for-report: s-request
       (referee.code,report-submitted)
     set-no-waits: no-wait-reports<--1
  end;
```

The "all-refs-agree" condition of the first transition is a compound Boolean expression which returns true if all referee script instances monitoring a particular author's paper are currently ready to execute a give command with the editor script and false otherwise. Note that the giving predicate only returns true or false depending on whether or not the named script instance has a matching give command waiting; it does not do the actual data transfer. For this, we must execute a set of corresponding give and take commands. That is

take((rflt.referee,author,editor,paper).
    referee-net,reply:YES)

in the "take-replies" property of the "referees-all-agree" transition corresponds with the "tell-editor-net" property

give((editor,author,paper).editor-net,
    ref-reply)

in the "referee-accepts" transition. Execution of these two commands cause the value of ref-reply in the referee script to be assigned to reply in the editor script (provided ref-reply has a value of type YES). The exit statement of "a22" is an efficiency consideration only. The first time the giving predicate of "a2" evaluates to false the "all-refs-agree?" compound condition is guaranteed to evaluate to false, regardless of whether or not all other unchecked referee script instances are waiting with corresponding give commands. By exiting "all-refs-agree?" as soon as its value can be determined, redundant checking of all other script instances for corresponding give commands can be avoided.

It should be noted that the construction of the "all-refs-agree" condition of the "referees-all-agree" transition means that all referee script instances must have a corresponding give command waiting for the condition to be evaluated to true. No particular reason exists for doing things this way; in fact

inter-script commands can be used to model just about any type of coordination requirement. Corresponding give and take commands are used to model similar types of coordination requirements between the "referee-refusal" and the "referee-refuses" transitions and between the "all-reports-in" and "report-made" transitions in the EDITOR-NET and REFEREE-NET scripts.

4.3 Generalization Hierarchies for Scripts

As a simple example of the use of generalization in organizing scripts, consider specializing the "have-referees" transition of EDITOR-NET to include a new condition ensuring at least 2 referees with expertise in the paper's area have been designated for all scientific papers.

```
transition have-referees on
  (EDITOR,AUTHOR,SCIENCE-PAPER)..editor-net
is
  conditions
    at-least-2-refs?: (begin
        locals
          count:INTEGER
        actions
          a1: count <-- 0
          a2: for rflt in SCIENCE-REFLIST
              where ((rflt.editor=editor)
                and (rflt.author=author)
                and (rflt.paper=paper)
                and (paper.area instance-of
                    rflt.referee.area))do
              count <-- count + 1
        returns
          rtn: (count >= 2)
      end) = true
```

This new condition checks that two or more referees have been assigned to review the paper by counting the number of appropriate instances in SCIENCE-REFLIST. The "paper.area instance-of rflt.referee.area" condition ensures that the area of the paper matches one of the referee's areas. The "area" property of REFEREE has a set of tokens as its p-value. Of course, the "all-refs-in?" condition is inherited intact and both conditions must now be true before the transition can be executed and fired in the specialized script.

A more interesting specialization of EDITOR-NET would require that the editor be informed of all referees late with their reports. This can be accomplished by adding a new transition to EDITOR-NET as follows:

```
transition ref-reports-late on
  (SCIENCE-EDITOR,SCIENCE-AUTHOR,
   PAPER)..editor-net is
  input from:waiting-report
  output to:waiting-report
  conditions
    time-exceeded?: (current-time -
      report-time) > "2 weeks"
  actions
    tell-ed-rpts:
      for rflt in SCIENCE-REFLIST
      where ((rflt.editor = editor)
        and (rflt.author = author)
        and (rflt.paper = paper))do
      if not (giving ((rflt.referee,
        editor,author,paper).referee-net,
        report-in:TRUE) then
      s-inform (editor.code,rflt.referee,
        rflt.author,rflt.paper,
        "report late")
    wait-again:report-time<--current-time
end
```

This transition is an example of a timing event in that every 2 weeks the editor receives a list of all referee information for referees who have not yet submitted their report. (He is responsible for taking some action to resolve each referee's tardiness). This list is compiled by checking for all appropriate REFEREE-NET instances that are not waiting to communicate a value of true to EDITOR-NET. The "report-time" local variable has to be initialized to the current time before the "ref-reports-late" transition first becomes enabled. This is done by specializing the preceeding "referees-all-agree" transition as follows:

```
action set-wait-time on
  (SCIENCE-EDITOR,AUTHOR,SCIENCE-PAPER)..
   editor-net..referees-all-agree is
   report-time <-- current-time
```

Now, whenever the "waiting-reports" state is marked (as a result of the "referees-all-agree transition firing) both the "all-reports-in" and the new "ref-reports-late" transition will be enabled. The "ref-reports-late" transition will continually execute and fire at 2 week intervals until the "all-reports-in" transition is executed and fired; in which case "ref-reports-late" becomes disabled.

5 Concluding Remarks

We have described in this paper an extension to Taxis which allows the designer of an IIS to describe within one conceptual framework a database, the application programs that operate on it, and the dialogues to be supported by the IIS for I/O communication with users. Our proposal assumes that software development costs will be reduced, not by decoupling the design of user interfaces from that of the database and its application programs, but by offering design tools which allow a integrated design of all components of an IIS, from its database to its user interfaces.

The script mechanism described here has served as a basis for the design of two medical information systems: one involving a pacemaker database <DiMarco 82> and the other a clinical trial management system <Buchan 82>. Although some revisions had to be made to the script mechanism, it was generally found to be extremely useful in both designs. A version of the script mechanism is currently being implemented.

Bibliography

<Barron 80>

Barron J.L., "Dialogue Organization and Structure for Interactive Information Systems", Technical Report CSRG-108, Dept. of Computer Science, University of Toronto, Feb., 1980.

<Barron 81>

Barron J.L., "Dialogue and Process Design for Interactive Information Systems Using Taxis", Technical Report CSRG-128, Dept. of Computer Science, University of Toronto, April, 1981.

<Borgida et al 81>

Borgida A.T., Mylopoulos J., and H.K.T. Wong, "Methodologies and Computer Aids For Interactive Information System Development", IFIP Working Conference on Automated Tools for Information Systems, Design and Development, New Orleans, Jan., 1982.

<Buchan 82>

Buchan I., "Clinical Trial Management Systems and Taxis", forthcoming MSc Thesis, Dept. of Computer Science,

University of Toronto, 1982.

<Di Marco 82>

Di Marco C., "Design of a Medical Information System Using Taxis", forthcoming MSc Thesis, Dept. of Computer Science, University of Toronto, 1982.

<Hoare 78>

Hoare C.A.R., "Communicating Seguential Processes", CACM, Aug. 1978, Vol.21, No.8, Pages 666-677.

<Mylopoulos et al 80>

Mylopoulos J., Bernstein P.A., and H.K.T. Wong, "A Language Facility for Designing Interactive Database-Intensive Applications", ACM TODS, Vol.5, No.2, June, 1980.

<Peterson 77>

Peterson J.L., "Petri Nets", Computing Surveys, Vol.9, No.3, Sept., 1977, Pages 223-252.

<Pilote 82>

Pilote M., "The Design of User Interfaces for Interactive Information Systems", PhD Thesis (in preparation), Dept. of Computer Science, University of Toronto, 1982.

<Smith and Smith 77>

Smith J.M. and D.C.P. Smith, "Data Base Abstractions", ACM TODS, Vol.2, No.2, June, 1977, pp105-133.

<Wasserman 77>

Wasserman A.I., "Procedure-Oriented Exception Handling", Technical Report #27, Medical Information Science, University of California, San Francisco, Feb., 1977.

<Wirth 71>

Wirth N., "Program Development by Step-wise Refinement", CACM, Vol.14, No.4, April 1971, Pages 221-227.

<Wong 81>

Wong H.K.T., "Design and Verification of Interactive Information Systems", Ph.D. Thesis, Dept. of Computer Science, University of Toronto, 1981.

<Zisman 77>

Zisman M.D., "Representation, Specification, and Autcmation of Office Procedures", Ph.D. Thesis, Dept. of Decision Science, The Wharton School, University of Pennsylvania, Sept., 1977.

Office Information Models and the Representation of 'Office Objects'

Simon J. Gibbs

Computer Systems Research Group
University of Toronto
Toronto, Ontario
Canada

Abstract Office information models are used to represent the operations and information structure of Office Information Systems. This paper determines requirements for these models by examining the functionality of Office Information Systems. The paper then discusses the representation of 'office objects' and their associated operations of filing, mailing and formatting. This representation is based upon concepts from the field of semantic data modelling.

1. Introduction

Computer-based Office Information Systems (OIS's) are concerned with the representation and management of information relevant to offices. Here 'information' is used in the sense of *interpreted* data; the OIS not only stores raw data but also provides facilities for the interpretation of this data by the end user.

The logical structure of data is specified by means of a data model. Data models also support primitive operations for manipulating data. For example, the three traditional data models, the network, hierarchical and relational models, organize data using networks, trees and tables respectively. Primitive operations include such things as updating and retrieving units of data. In order to describe the information content of a system we will make use of what may be termed an *information model*. Such a model, in addition to specifying data structure and primitive operations, contains constructs which aid in the interpretation of data.

An *office information model* is an information model designed specifically for the office domain. The second section of this paper discusses the role of these models in relation to the OIS and identifies a number of requirements for such models. The third section shows how various concepts from the field of semantic data modelling may be extended, providing a basis for the representation of 'office objects' by office information models.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0021 $00.75

2. Office Information Models

Two of the major problems facing the development of Office Information Systems are integration and the design of suitable user interfaces [5, 16]. Both problems stem from the diversity of OIS functionality and the associated tendency to divide the OIS into numerous semi-independent subsystems. This approach discourages integration since each subsystem, such as electronic mail, information retrieval, document preparation etc., may have its own representational peculiarities which complicate the exchange of information between subsystems. In addition the user interface may become complex as each subsystem will phrase its operations in a possibly different command language.

Office information models address the problems of integration and interface design by providing a single data representation and a uniform set of operations. The particular model described in this paper supports the representation of 'office objects'. These are the information bearing objects found in offices; examples are forms, letters, dossiers and so on. The use of interpretative information within the model allows the description of an object's structure at two levels. The first is data oriented, suitable for the specification of operations. The second level is information oriented and intended for user interaction. We will borrow the terms *datalogical* and *infological* [10] to distinguish these two levels.

Before introducing the model let us consider some of the requirements of office information models in general. We may divide these requirements into those for the datalogical level and those for the infological level. At the datalogical level we will require that:

- The model must support the different types of data found within the office.
- The model must support some aspect of communication.

These two requirements arise from the OIS functions of information management (storage, acquisition and retrieval) and communication [16]. Although Office Information Systems may provide a number of other functions, such as decision support or calendar management, the information management and communication functions are in some sense minimal; it is difficult to classify as an OIS a system which does not support these functions.

The first of the above requirements refers to the various types of data used within the office. These include strictly formatted numeric and character data such as found in tables or forms, text and graphic data such as found in documents, and audio data such as may be found in messages. Most current data models treat only strictly formatted data.

However, the use of non-strictly formatted data is widespread within the office. Any system which attempts to support office activities must take this into account.

The second requirement derives from the distributed configuration of Office Information Systems. One may view an OIS as a network of information bearing stations. A very common operation within the OIS is the transfer of information from one station to another. An office information model should support the station concept and allow information transfer between stations.

Requirements for the infological level are related to the human factors issues of data model design [12]. Specifically, we will require that:

- The model should make use of natural structure.
- The model should be easily extended.

While the above two requirements are desirable in most computer systems, they merit repeating since ease of use and flexibility are crucial to the acceptance of Office Information Systems.

The requirement for natural structure, i.e., that perceived by the user, allows the data representation used by the office information model to be based upon concepts familiar to the user. This aids the user in interpreting data and performing operations within the OIS.

The extensibility requirement is particularly important for Office Information Systems. The office is a dynamic environment in which procedural and structural changes are to be expected. The OIS must be easily extended to reflect these changes as they occur.

The field of semantic data modelling [2, 3, 8, 15, 17, 20, 21] has long been concerned with the questions of natural structure and extensibility. Semantic data models thus provide a good starting point for the design of an office information model. However, these models lack the data types and communication mechanism required by Office Information Systems. The following section discusses how concepts from semantic data modelling may be extended to the office domain. A rudimentary office information model, and its representation of filing, mailing and formatting, is described.

3. Office Object Representation

Semantic data models often make use of abstract objects which correspond directly to objects in the real world. The analogous concept in office information models is that of *office object*. An office object is any information found in the office which we wish to be individually represented within the model. For example, we could choose to represent documents as office objects. In this case such things as sections of documents or the pages of a document cannot themselves become office objects. This does not imply that we cannot represent the internal structure of documents, merely that documents are to be taken as indivisible or primary objects.

An object* is described by its properties and their values. A property may be *single valued* or *multivalued*, in the latter case the value of the property for a given object may be a set. For instance, an object such as an application form may have a property used to record the telephone number of the applicant. If the applicant is only allowed to give one telephone number then the property is single valued. If the applicant may give more than one telephone number, the property must be multivalued.

Properties are either *composite* or *primitive*. A primitive property has an underlying data type. The allowable data types should include boolean, numeric, character, graphic and audio. The first three data types are those most commonly found in current data models. Boolean and numeric properties are useful for describing strictly formatted objects such as forms. Character properties may also be used with forms and any object which contains text. The graphic data type is used for digitized images, the audio data type for digitized voice. It is assumed that there is no selective retrieval based upon audio or graphic property values. However, graphic or audio information may be retrieved by using associated character, numeric or boolean information. This is similar to the use of keywords as a basis for text retrieval.

Composite properties are formed from a number of *sub-properties* which may in turn be either composite or primitive. The value of a composite property is the cross product of the values of its sub-properties. For example, the composite property **address** could be formed from the primitive sub-properties **province**, **city**, **street** and **street-no**. A typical value for this property could be *(Ontario, Toronto, Beverley, 210)*.

Collection of objects with similar properties are known as *object types*. Each member of an object type is said to be an *instance* of the object type. An object type is defined by specifying the properties of its instances. For example, the object type **order-form** may be defined as:

order-form --> **date item+ total**
item --> **quantity part-no price**
date --> character
total --> numeric
quantity --> numeric
part-no --> numeric
price --> numeric

An **order-form** has three properties: **date**, **item** and **total**. The properties **date** and **total** are primitive, using the character and numeric data types. The **item** property is composite, consisting of the three primitive properties **quantity**, **part-no** and **price**, each of which is numeric. In addition the **item** property is multivalued, this is indicated by the '+' in the object type definition. An instance of the object type **order-form** is shown in Figure 1.

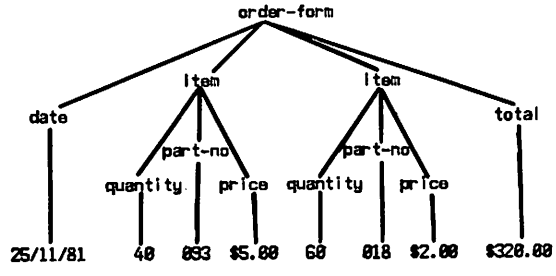


Figure 1 An order-form instance.

The specification of a property's data type imposes a constraint on the physical representation of the property's values. The exact physical representations of the various data types need not concern us here. It is assumed that once the system knows the data type of a property value it can process the value accordingly.

* Office objects will simply be referred to as objects when the meaning is clear from context.

Semantic constraints on property values are also necessary. For example, the **part-no** property of **order-form** must only be assigned valid part numbers. Also certain property values may be derived from the values of other properties. Here **order-form** again provides an example, since **total** should equal **quantity** times **price**. However, a discussion of the specification of semantic constraints and derived properties is outside the scope of this paper.

As a second example of object type definition consider:

letter --> **sender receptant body date**
body --> **paragraph+**
paragraph --> **sentence+**
sender --> character
receptant --> character
date --> character
sentence --> character

The **letter** object type is a simple text document. In general text documents are hierarchically structured. Here **letter** contains a **body** with a variable number of paragraphs, a paragraph may be further decomposed into a variable number of sentences (see Figure 2). Hierarchically structured documents have recently been used in a number of text composition systems [1, 7, 19]. Document structure may be used to specify editing operations [7]. For example, the operation which removes a sentence from a document can be easily visualized as a reorganization of the underlying hierarchy. In addition document structure is useful during retrieval [22]. Thus, the user may ask to see the structural skeleton of a document or to view a particular structural component.

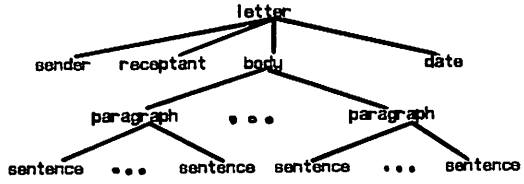


Figure 2 A letter instance (property values not shown).

A final example of an object type definition is:

employee-record --> **sin name birthdate salary photo**
sin --> numeric
name --> character
birthdate --> character
salary --> numeric
photo --> graphic

The **employee-record** object type contains a graphic valued property, here used for an employee's photograph. An instance of the **employee-record** object type is shown in Figure 3. Queries with selection conditions on **photo** values are not allowed. However, it would be possible to retrieve particular **photo** values by specifying associated information such as **sin** (social insurance number) or **name** values.

Figures 1 to 3 illustrate the hierarchical structure of the properties of office objects. Hierarchical organization of properties is found in a number of semantic data models [3, 20, 21] and has also been used in office oriented data models [13, 14]. An advantage of this organization is that it lets us view the properties of objects at different levels of detail. Furthermore, similar organization may be found in natural language, where objects (nouns) may be described by property hierarchies [21].

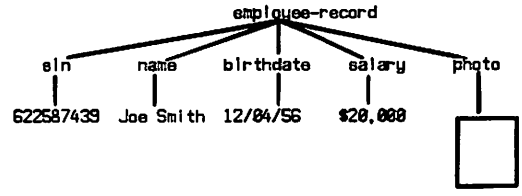


Figure 3 An employee-record instance.

Different object types within the office may be related by the *is-a* relationship [17]. Given object types **X** and **Y**, we say **X** *is-a* **Y** if every instance of **X** is also an instance of **Y**. The object type **X** is said to be a *specialization* of **Y** (conversely **Y** is a *generalization* of **X**) and may contain additional properties to those of **Y**. For example, a specialization of **letter** could be **letter-of-reference**. Specialization and generalization are commonly used in semantic data models [15] and also object oriented systems such as Smalltalk [6] and SBA [4]. An advantage of using *is-a* is that the system may be easily extended by specializing previously defined types.

The various *is-a* relationships that exist between object types may be depicted by a graph with object types as nodes and arcs corresponding to *is-a* relationships. Although in general this graph may be a network, it is often referred to as an *is-a* hierarchy. The *is-a* hierarchy for office objects is shown in Figure 4 (here the direction of increasing specialization is downwards). The most general type, *office-object*, has all office objects as its instances.

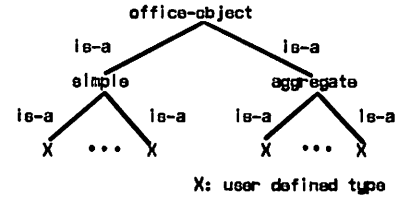


Figure 4 The is-a object type hierarchy.

Each object type may be further characterized as *simple* or *aggregate*. Aggregate objects may 'contain' other objects. Containment is expressed using the *part-of* relationship. For example, given the aggregate object x (i.e., x is an instance of an aggregate object type) and some object y, then we say y *part-of* x if x contains y. Simple objects are those not allowed to appear on the right hand side of the *part-of* relationship. However, simple objects can be contained within aggregate objects. This method of aggregation is similar to the 'cover-aggregation' of RM/T [3]. Here though, *part-of*, rather than type membership, is used to relate the constituents of an aggregate instance. Consequently aggregate types are not elevated to the level of metaclasses and can be directly included within the *is-a* hierarchy.

A natural operation on an aggregate object is to remove a constituent and include the constituent within a second aggregate. When this occurs we will say that the first object is 'put' into the second object or that the first object has 'moved'. Depending upon the aggregate objects used, this operation may represent such diverse user activities as filing, mailing and formatting. In the following, common aggregate object types are described.

3.1 Files and Dossiers

The definition of an aggregate object type can include constraints on the allowable constituents of the aggregate's instances. A *file* is an aggregate object type for which the constituents of all instances belong to a single type. For example, we can define an aggregate object type named **order-file** and specify that the constituents of instances of **order-file** be of type **order-form**. If y is an instance of **order-file** and x *part-of* y holds, then x must be an instance of **order-form**. Alternatively, we may define an aggregate object type with no constraints on constituents. Such objects are called *dossiers*. For example, let d be an instance of the aggregate type **personnel-dossier**, x an instance of the object type **job-application**, and y an instance of the object type **resume**. It is not necessary for x and y to be of the same type if they are both constituents of d, i.e., both x *part-of* d and y *part-of* d are possible. The *part-of* relationship allows one aggregate object to be *part-of* another. So, if $d1$ and $d2$ are both dossier instances then $d1$ *part-of* $d2$ is also possible.

Properties of an aggregate object can be used for descriptive or summary information about its constituents. Thus a property of **order-file** could be **number-of-orders**, where for each instance f of **order-file** the value of **number-of-orders** is equal to the number of constituents of f. It is not necessary for the property values of an aggregate object to uniquely identify the object since the object's constituents can be used to differentiate between instances.

The filing of an object is represented by including the object as a constituent of a file or dossier instance. Two methods of retrieval are possible. First, by viewing the object as an instance of a type and specifying a selection condition involving the type's properties. Second, by viewing the object as a constituent of an aggregate. If the aggregate is homogeneous, as in the case of files, then again selection conditions based on property values may be given. For nonhomogeneous aggregates, i.e., dossiers, more general conditions, such as pattern matching, may be specified.

3.2 Stations

The *part-of* relationship imposes a hierarchy on object instances. The root of a *part-of* hierarchy is always an instance of the **station** aggregate object type. The purpose of **station** is to group together all objects which belong to a single station within the OIS. In a distributed configuration it will be assumed that all constituent objects of a **station** instance are located on the same node. Transfer of an object from one station to another is represented by removing the object from the *part-of* hierarchy of the first station and placing it under that of the second station. Rather than having the user move the object directly to the second station it is convenient to introduce the **in-tray** and **out-tray** aggregate object types. Typically a station contains both an **in-tray** and an **out-tray** instance. When an object is put into an out-tray the OIS automatically transfers the object to the in-tray of the destination station.

There is a problem, though, in associating routing information with the constituents of in-trays and out-trays. Consider the in-tray as an example, somehow, in order to perform the transfer, the OIS must know the destination of each constituent. Assigning **in-tray** a multivalued property for use as a destination list is not possible since there is no way to match up the property's values with constituents. An alternative is to allow each in-tray only a single constituent at a time, in this case **in-tray** could have a single valued property used to record the destination. However, this could frustrate the user, particularly if there are delays in object transfer. A second alternative is to require that each object type have special properties which are reserved for routing information. This is unsatisfactory though, since changes in the mailing system potentially affect all previously defined object types. A more realistic solution makes use of the **envelope** aggregate object type. Among the properties of **envelope** are **to** and **from** which are used to identify the destination and source stations. Additional properties could be used to specify such things as service class, if the OIS provides different classes of service, or perhaps security information.

Suppose the user of station $s1$ wishes to transfer the object x to station $s2$. (It is not necessary for x to be a simple object, if x is an aggregate then the constituents of x are transferred also.) The steps performed by the user are as follows: First an **envelope** instance, e, is created. In doing so the user specifies the destination station $s2$. Next the user puts x into e and e into $o1$, the out-tray of station $s1$ (see Figure 5). The remaining steps of the transfer are performed by the OIS. This example illustrates how closely operations within the model resemble those that would be performed in a paper-based system.

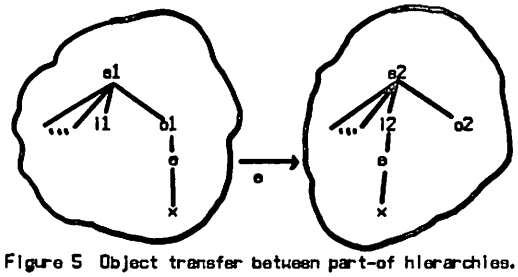


Figure 5 Object transfer between part-of hierarchies.

In order for a user to 'mail' an object to a second user it is only necessary to know the object to be transferred and the destination station. These two objects may be selected by the general method of specifying selection conditions on property values (it would be useful for **station** to have a property identifying the users of particular **station** instances).

3.3 Templates

Until now we have been considering objects and their properties at the datalogical level. The transformation to the infological level involves the use of *templates* [23]. A template is a structure which provides an interpretation for an object. Suppose x is an object and $\mathbf{T}$ a template. We may symbolically represent the interpretation of x under $\mathbf{T}$ as $\mathbf{T}(x)$. The purpose of the transformation $\mathbf{T}(x)$ is to present x to the user in an understandable form. For example, instead of presenting the user with a simple list of x's property values, we may wish to provide additional information such as how the property values are related and what operations may be performed on these values. Of course it cannot be guaranteed that $\mathbf{T}(x)$ is any more understandable than x itself. The success with which $\mathbf{T}$ provides adequate interpretations depends upon the definition of $\mathbf{T}$.

A *template type* is an aggregate object type used for representing templates. Let $\mathbf{T}$ be a template type and x an arbitrary object. If x *part-of* t holds with t an instance of $\mathbf{T}$, then we say $\mathbf{T}(x) = t$. So, in order to interpret an object it is only necessary to include the object as a constituent of some template instance. As with other aggregate objects, constraints may be specified which determine the allowable constituents. Viewing $\mathbf{T}$ as a function, these constraints determine the domain of $\mathbf{T}$.

With the exception of audio properties, property values are visually presented to the user. A template describes the positioning of property values in the visual field. In general, a template definition consists of:

1) boxes (which may be nested),
2) positioning constraints between boxes,
3) a mapping mechanism which associates property values with boxes,
4) a specification of the allowed operations upon the contents of the various boxes,
5) 'background' information,
6) an icon.

Specific regions within the visual field are referred to as *boxes* [7]. A template is a hierarchy of boxes. Property values are displayed within boxes, as is any 'background' information (this is information which is the same for all instances of a template type). Each box has a number of operations which can be used to manipulate the contents of the box. A graphic symbol known as an *icon* can be used to identify the template type.

Examples of templates are shown in Figures 6 to 9. The **order-template** (Figure 6) would be used to display instances of type **order-form**. The positions for the **date**, **total** and **item** property values are indicated on the template. Similarly Figures 7 and 8 show templates to be used with **letter** and **employee-record** objects. The operations associated with the various boxes of these templates are not shown, but would include such things as editing and scrolling. When the user selects a particular box, a menu of the available operations can be displayed. In some cases it is useful to explicitly show the operations associated with the box. Consider the template in Figure 9, which is intended to be used with the following object type:

voice-message --> **to from date**
subject message reply
to --> character
from --> character
date --> character
subject --> character
message --> audio
reply --> audio

Because of the ephemeral nature of sound, it is difficult to know just when to output audio property values. Here the decision is left to the user. In **voice-message-template** the boxes which correspond to the audio valued properties **message** and **reply** are menus. By selecting a menu command the user can control the output of the associated audio property value.

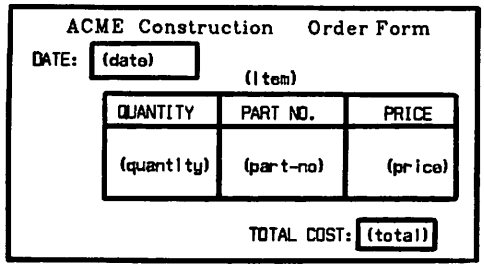


Figure 6 The order-template.

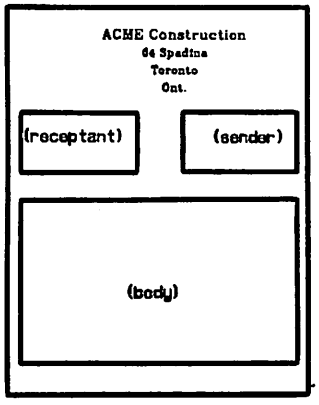


Figure 7 The letter-template.

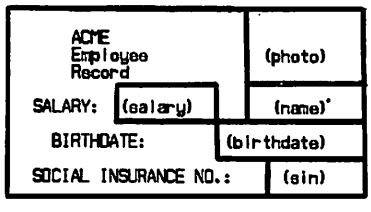


Figure 8 The employee-record-template.

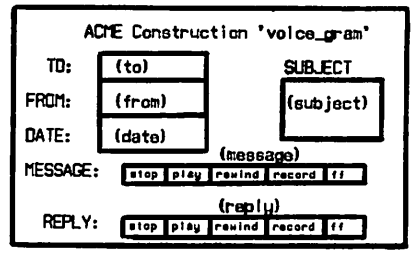


Figure 9 The voice-message-template.

Often it is useful to identify the type of an object without displaying the object in all its detail. For this purpose each template has a graphic valued property used for an icon. For example, the icon for **envelope** could be a small rectangle with a 'stamp' on it, the icon for a file aggregate type could resemble a file cabinet. The operations which can be performed upon an icon are those associated with the outermost box of the template. These operations include such things as displaying the complete template and moving the object contained within the template. This use of icons is similar to that found in the Star system [18] and spatial data management systems [9].

Templates have many uses in addition to formatting and the support of user interaction. One such additional function is to provide different views of an object: given templates **T1** and **T2** and an object x, then **T1**(x) may be quite different from **T2**(x). For example, two separate departments may use different templates for the same form type. Templates have also been used in the specification of queries [24, 25] and office procedures [24].

4. Conclusion

This paper has outlined an office information model and discussed the representation of objects and operations within the model. The model has a number of interesting features: First, data values are not restricted to the standard boolean, character and numeric types; the model allows audio and graphic data. Secondly, a number of concepts from the

field of semantic data modelling have been used. These include the abstraction mechanisms of aggregation and generalization and the use of objects. A similar approach has previously been suggested for the design of end user facilities [11]. The approach used in this paper is to represent objects such as stations and files within the model itself. Finally, the model contains interpretative constructs, known as templates, which aid the user in understanding the system.

The objects and operations described in this paper are relatively simple. More complex aspects of the office, such as office procedures, involve operations and constraints which are difficult to formulate within the model as they make use of application specific knowledge. One possible direction for future work is to try and include such knowledge in the model.

References

[1] Chamberlin, D.D. et. al. JANUS: An interactive system for document composition. Proc. ACM Symposium on Text Manipulation, Portland, Oregon, June 1981, pp. 82-91.

[2] Chen, P.P.S. The entity-relationship model: Toward a unified view of data. ACM Trans. Database Syst. *1*, 1 (March 1976), 9-36.

[3] Codd, E.F. Extending the database relational model. ACM Trans. Database Syst. *4*, 4 (Dec. 1979), 397-434.

[4] deJong, P. The system for business automation (SBA): A unified application development system. Proc. 1980 IFIP Congress, pp. 469-474.

[5] Ellis, C., and Nutt, G. Computer science and office automation. Comput. Surv. *12*, 1 (March 1980), 27-60.

[6] Goldberg, A., and Kay, A. Smalltalk-72 instruction manual. XEROX Palo Alto Research Centre Report, March 1976

[7] Hammer, M., et. al. The implementation of Etude, an integrated and interactive document production system. Proc. ACM Symposium on Text Manipulation, Portland, Oregon, June 1981, pp. 137-146.

[8] Hammer, M., and McLeod, D. Database description with SDM: A semantic database model. ACM Trans. Database Syst. *6*, 3 (Sept. 1981), 351-386.

[9] Herot, C.F. Spatial management of data. ACM Trans. Database Syst. *5*, 4 (Dec. 1980), 493-513.

[10] Langefors, B. Infological models and information user views. Inform. Systems *5*, 1 (1980), 17-32.

[11] Lefkovits, H.C., et. al. A status report on the activities of the CODASYL end user facilities committee (EUFC). ACM SIGMOD Rec. *10*, 2 & 3 (Aug. 1979), 1-28.

[12] Lochovsky, H.V. Human factors issues in office information systems. Proc. International Congress on Applied Systems Research and Cybernetics, Acapulco, Mexico, Dec. 1980.

[13] Lum, V.Y., Shu, N.C., Tung, F., and Chang, C.L. Automating business procedures with form processing. IBM Res. Rep. RJ3050, March 1981.

[14] Luo, D., and Yao, S.B. Form operation by example: A language for office information processing. Proc. 1981 ACM SIGMOD Conf., pp. 212-223.

[15] McLeod, D., and Smith, J.M. Abstraction in databases. Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, Pingree Park, Colorado, June 1980.

[16] Morgan, H.L. Research and practice in office automation. Proc. 1980 IFIP Congress, pp. 783-789.

[17] Mylopoulos, J., Bernstein, P.A., and Wong, H.K.T. TAXIS: A language facility for designing database-intensive applications. ACM Trans. Database Syst. *5*, 2 (June 1980), 185-207.

[18] Purvy, R., Farrell, J., and Klose, P. Star's records processing: Data processing for non-computer professionals. Proc. 1982 ACM SIGOA Conf. on Office Information Systems.

[19] Reid, B.K. A high-level approach to computer document formatting. Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nev. (Jan. 1980), pp. 24-31.

[20] Schmid, H.A., and Swenson, J.R. On the semantics of the relational data model. Proc. 1975 ACM SIGMOD Conf., pp. 211-223.

[21] Smith, J.M., and Smith, D.C.P. Database abstractions: Aggregation. Commun. ACM *20*, 6 (June 1977), 405-413.

[22] Stromfors, O., and Jonesjo, L. The implementation and experiences of a structure-oriented text editor. Proc. ACM Symposium on Text Manipulation, Portland, Oregon, June 1981, pp. 22-27.

[23] Tsichritzis, D. Form management. Submitted for publication.

[24] Tsichritzis, D., Rabitti, F., Gibbs, S., Nierstrasz, O., and Hogg, J. A system for managing structured messages. IEEE Tran. on Commun. *Com-30*, 1 (Jan. 1982), 66-73.

[25] Zloof, M.M. Query-by-example: A database language. IBM Systems J. *16*, 4 (1977), 324-343.

AN OIS MODEL FOR INTERNAL CONTROL EVALUATION

Andrew D. Bailey, Jr.
University of Minnesota

James Gerlach
Andrew B. Whinston
Purdue University

R. Preston McAfee
University of Western Ontario

INTRODUCTION

Satisfactory accountability, that is, a control structure which protects corporate assets from theft, misuse, and fraud, is an important aspect of office information system design. The OIS design goals of flexibility, efficiency, and modularity must not preclude the accountability needs of managers, stockholders, and auditors. This will require meticulous examination of the OIS to ensure that it satisfies the multi-attribute control criteria used by auditors.

Currently, requirements are met by employing traditional flowchart descriptions, questionnaires with a focus on potential weaknesses and exposures, narrative descriptions, decision tables, and other methods that seem appropriate to the individual auditor in each situation. The review and evaluation stage of this process is largely limited to human analytic capabilities, which are surprisingly deep, broad, and complex - but extremely varied and unreliable.

New techniques are needed to support the auditor in the review and evaluation process. Many of these techniques will take advantage of computer technology and its capacity to deal swiftly and accurately with highly complex systems of relationships. Our approach is to model the firm's internal behavior, of which the OIS is a subset, using the computer-acceptable Internal Control Description Language. The model is then analyzed by machine to see that it satisfies auditing criteria. In this way, our system - called TICOM-II - incorporates both OIS theory and auditing criteria.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0027 $00.75

OVERVIEW

The TICOM-II modeling and analysis approach to unifying OIS and auditing issues has four distinct components. The first, the Internal Control Description Language, is a modeling language for formally describing a firm's operations. This formal model is then mapped algorithmically to an internal representation, the second component. The internal representation was designed to facilitate analysis, the third major component of TICOM-II. The analysis methods are controlled by the fourth component, a query processing system that permits the auditors to pose questions concerning the internal control model's behavior. For more information concerning the the Internal Control Description Language and TICOM-II, consult Bailey, et. al. [1, 2, 3, 4].

INTERNAL REPRESENTATION

The OIS model employed in TICOM-II is a precedence oriented model which incorporates principles taken from first order logic. Precedence-oriented models depict the office as a set of tasks whose permissible execution sequences are specified by precedence constraints. The general form of the model is a bilogic directed graph showing both control and data flow. Each node has a precedence relationship with other nodes that is described by listing the nodes immediate predecessors and the conditions that must be satisfied if the node is to follow its immediate predecessors. The action of each node is described in terms of variable assignments.

ANALYSIS

The types of questions posed by auditors during internal control evaluation are diverse, many of the questions concern state achievability. That is, is it possible for the firm, given its internal control structure, to reach a particular state. If it is possible to enter such a state,

TICOM-II analysis establishes a precondition for entering that state and a postcondition that is necessarily true when the state is achieved. The former permits analysis of the strength of safeguards in the system, while the latter allows identification of what to examine in regards to identifying the perpetrator when a control circumvention is suspected. Precondition evaluation is based on and consistent with Dijkstra's notion of weakest precondition [5].

Precondition and postcondition evaluation of accounting models is closely related to the analysis of programs. As such, automated accounting internal control evaluation suffers from many of the same ills that cripples program verification [6]. To combat these problems several measures have been taken.

Office activity is modeled at a high-level of abstraction which focuses on the major details of the system. Details of office procedures are suppressed in favor of a simpler firm-wide perspective. Once the internal control evaluator understands the sequencing of office procedures, detailed examination of the system can proceed on an individual office procedure basis. In addition, TICOM-II has a system simplification facility for creating "black boxes". A "black box" is a simplified component of the model in which all but the most essential features are suppressed. Two popular approaches to internal control evaluation, the cycles approach and the account classification approach, are based on this technique. Finally, the modeling and analysis of parallelism is restricted to noninterfering office activities. That is, if two or more office activities can be performed in parallel all permissible execution sequences terminate with identical results. The reasonableness of this restriction is argued from an accounting perspective.

CONCLUSION

Verification of an accounting internal control system is an NP-hard problem [7]. Yet auditors are required to review and evaluate them. It is our belief that this process can be effectively aided by computer-assisted tools such as TICOM-II and formal modeling and analysis is the first step towards the necessary imposition of accounting internal controls on an operating OIS.

REFERENCES

1. A. D. Bailey, Jr., James Gerlach, R. P. McAfee, and A. B. Whinston, "Office Automation," Handbook of Industrial Engineering, G. Salvendy ed. (Wiley & Sons, Inc., NY, forthcoming).

2. _____, "Internal Accounting Controls in the Office of the Future," IEEE Computer, May 1981, pp. 59-70.

3. _____, "TICOM-II - The Internal Control Language," Florida Symposium Internal Controls, ed. by A. Rashad Abdel-Khalik, forthcoming.

4. _____, "An Introduction to the Theoretic and Analytic Capabilities of TICOM-II," Proc. of the 2nd Intn'l Workshop on OIS, IRIA, October 1981.

5. E. W. Dijkstra, "Guarded Commands, Nondeterminancy and Formal Description of Programs," CACM 18, 453-457, August 1975.

6. R. A. Demillo, R. J. Lipton, A. J. Perlis, "Social Processes and Proofs of Theorems and Programs," CACM 22, 271-280.

7. A. D. Bailey, Jr., R. P. McAfee, and A. B. Whinston, "Application of Complexity Theory to the Analysis of Internal Control Systems," Auditing: A Journal of Practice and Theory, Vol. 1, No. 1, Summer 1981, pp. 38-52.

How Do People Really Use Text Editors?

John Whiteside, Norman Archer, Dennis Wixon, and Michael Good

Digital Equipment Corporation
Maynard, Massachusetts 01754

Abstract

Keystroke statistics were collected on editing systems while people performed their normal work. Knowledge workers used an experimental editor, and secretaries used a word processor. Results show a consistent picture of free use patterns in both settings. Of the total number of keystrokes, text entry accounted for approximately 1/2, cursor movement for about 1/4, deletion for about 1/8, and all other functions for the remaining 1/8. Analysis of keystroke transitions and editing states is also presented. Implications for past research, editor design, keyboard layout, and benchmark tests are discussed.

1. Introduction

Different approaches have been used to study usage of text editors. Several researchers have proposed models of user performance and then tested user behavior with respect to these models [1, 2, 3, 4, 5]. A second approach has been to compare editors by having subjects perform specific benchmark tasks [9]. A third approach has been to compare command transition matrices for various editors [6].

1.1. Ideal Models

Models of text editor performance have focused either on predicting the time which users would take to perform a particular task, or on the strategy which users would employ to accomplish a particular goal. Models which focus on time [1, 2, 3] treat the overall task time as the sum of the times which various subtasks take. These subtasks include operations such as typing, pointing, thinking, and waiting for system response. Models which focus on commands include the GOMS model [1], Hammer's constrained optimal model [5], and Embley and Nagy's model based on file comparison [3].

All of these models share a common attribute of concentrating on idealized performance. For example, Card et al. ignore errors, give subjects practice problems, and instruct them with respect to how particular activities are to be performed. Alternatively, the performance of users is considered as a deviation from the model. Such deviation is interpreted as suggesting the need for user training [5] or has been tabulated and classified according to user experience [3]. Hammer found that an optimal model predicted user performance rather poorly. Constraining the model by taking into account only those commands which were actually used improved the fit considerably.

1.2. Benchmark Tasks

One of the most detailed analyses of editors was conducted by Roberts [9]. She developed methods for comparing editors on the basis of speed of use, ease of learning, range of functionality, and error-proneness. Speed analyses of editors involved the performance of benchmark tasks by experts on each system. She used these methods to compare the TECO, Wylbur, NLS, and Wang text editors. TECO and Wylbur were found to be significantly slower to use than the other two editors. TECO was also significantly harder to learn than the other three editors.

1.3. Transition Matrices

Hammer and Rouse [6] analyzed free keystroke records for both programmers and documenters using either SOS or TECO. Behavior within an editor was categorized into uniform editor primitives, such as type one line, alter one character, type many lines, and alter many characters. The matrices of transition probabilities between editor primitives were then compared to an overall transition matrix

---

VAX, VMS, VT100, DEC, and PDP are trademarks of Digital Equipment Corporation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0029 $00.75

which represented the average of various editors. The results led Hammer and Rouse to conclude that the differences between editors could have been due to the fact that different people used them. In another analysis, Hammer and Rouse compared the same users editing programs and documents and found that about 75% of the subjects used similar techniques for each. All of these analyses were applied to editors which use hard copy terminals.

1.4. Present Approach

Instead of postulating an ideal editing model, we attempted to record and describe actual usage patterns. To this end keystroke-level data were collected while people performed their normal work on either an experimental editor or a word processing editor. Unlike previous research, this free-use logging was collected solely from screen editors and analyzed at several levels. Recommendations for editor design and evaluation procedures are based on actual editor usage.

2. Method

In the present experiment, users' keystrokes were logged and time-stamped for two editors. The first editor was an experimental editor called the Editor Prototyping Tool (EPT). The second editor was Digital Equipment Corporation's 200 Series commercial word processor (WPS). Both of these are character-oriented screen editors, but they differ in important ways.

2.1. EPT Description

EPT was designed specifically for research purposes. Thus, it contained a built-in keystroke-logging facility, designed not to influence software performance. EPT provided a simple set of editing functions including cursor movement anywhere on the screen, search, text relocation, and text deletion. Typing could be done in either insert or overstrike mode. These functions were activated by dedicated function keys; in most cases by a single keystroke. A short help facility was provided.

EPT contained few of the formatting functions present in a normal word processor; however it did perform automatic word wrapping.

2.2. WPS Description

The WPS word processor is a character-oriented, screen editor in which editing is done on the bottom line of the screen. The cursor may be moved within the boundaries of existing text. The keyboard has a variety of special keys for editing and cursor movement. A special "gold" key on the keypad, when used in combination with other keypad keys and with keys on the regular keyboard, greatly expands the number of possible functions which may be selected directly. WPS has logical equivalents for all EPT functions, and has many more functions in addition, including sophisticated text formatting functions.

2.3. Subjects

Six subjects contributed to the EPT work sample. These subjects were knowledge workers creating and revising their own reports and memos. The logging of data was not begun until all subjects had at least three weeks experience with EPT, which was sufficient time to allow these subjects to learn the features of this simple editor.

The subjects using WPS were eight secretaries who were carrying out regular secretarial and administrative duties for a research and development department of Digital Equipment Corporation. The experience of the subjects varied widely, with three having had almost two years of experience with the system and the full set of three training courses. One had a year experience and one training course. The remaining four subjects had no training course but were self taught, with experience of one to six months on the system.

2.4. Keystroke Logging

As mentioned above, EPT had a built-in logging facility. A permanent record of every keystroke was made, together with time of occurrence. EPT ran under the VMS timesharing system on a VAX-11/780 computer, supporting DEC VT100 terminals.

For the WPS word processing system, character streams and occurrence times from six VT100W terminals were collected by a PDP 11/34 computer, in such a way that WPS system performance was not affected. The recording was done over a period of 11 days.

3. Results

3.1. General Characteristics of the Work Samples

Table 1 shows some summary characteristics of the work samples.

	EPT	WPS
Total person-hours in sample	212	482
Hours taking breaks (over 150 sec)	127	390
Hours active work	86	92
Total keystrokes	510,503	406,102

Table 1: Comparison of EPT and WPS Work Samples

The samples contain roughly equal numbers of keystrokes, about half a million each. Total person hours in the WPS

Keys	Percent of keystrokes	Percent of time	Time per keystroke
Printing characters	53.68	46.84	0.53
Left-Arrow	12.90	5.08	0.24
Right-Arrow	12.52	5.49	0.27
Delete	7.35	7.38	0.61
Down-Arrow	5.67	9.89	1.06
Up-Arrow	3.15	5.89	1.14
Return	1.38	4.38	1.93
Tab	1.01	1.66	0.99
Word	0.34	0.39	0.71
Insert/Overstrike	0.33	1.02	1.86
Do It	0.28	0.97	2.14
Backup	0.24	0.54	1.36
Next Screen	0.22	2.34	6.38
Next Line	0.21	0.67	1.93
Advance	0.17	0.42	1.53
Exit	0.14	2.69	11.89
Last Screen	0.10	1.02	6.15
Erase	0.10	0.59	3.76
Search	0.07	0.66	6.17
Paragraph	0.05	0.17	2.23
Move	0.04	0.17	2.38
Cancel	0.03	0.28	5.92
Back Space	0.03	0.07	1.64
Copy	0.02	0.18	4.53
Goto Page	0.02	0.43	11.28
Sentence	0.02	0.43	12.80
Help	0.02	0.15	5.66
Command	0.01	0.13	7.74
Page	0.01	0.04	2.86
Totals	100.00	100.00	0.61

Table 2: EPT Keystroke Frequency Table

sample were over twice as many as in the EPT sample; this discrepancy was due to the secretaries' habit of turning on their machines in the morning and leaving them running for the entire day. EPT ran on a general timesharing system. so users were less likely to leave the editor running constantly. This difference can be seen by looking at the number of hours of inactivity (defined as the sum of all periods of greater than 150 seconds without a keystroke) which is much larger for the WPS sample. The remaining time, hours of active work, is comparable for both samples at somewhat under 100 hours.

3.2. Individual Keystroke-Level Analysis

Table 2 shows summary statistics for keystroke usage for the EPT editor. The left-hand column identifies the function. Each entry refers to an individual key, except for the case of "printing characters," which includes all alphanumerics and symbols under a single heading. The keys are ordered by frequency of usage, in terms of percentage of total keystrokes recorded. This number is shown in the second column; the percentages are based on the total number of keystrokes in the work sample (510,503). The remaining two columns in Table 2 show data with respect to time. The percentage of use by time is shown in the third column, and the mean time per keystroke is shown in the fourth column. In computing

Keys	Percent of keystrokes	Percent of time	Time per keystroke
Printing characters	34.81	47.48	0.71
Advance	16.48	11.91	0.63
Line	10.30	6.27	0.50
Backup	6.59	7.41	0.92
Rub Character	3.33	3.35	0.82
Return	2.83	8.08	2.33
Tab	1.52	1.79	0.96
Word	1.24	0.65	0.42
Delete Character	1.11	0.90	0.66
Paragraph	0.44	0.86	1.57
Rub Word	0.24	0.46	1.53
Menu	0.23	2.69	9.66
Page	0.21	0.97	3.71
Delete Word	0.19	0.19	0.83
Upper Case	0.17	0.11	0.61
File Document	0.12	1.61	10.50
Select	0.13	0.38	2.40
Ruler	0.12	0.60	4.18
Manual Pagination	0.08	0.78	8.24
Date & Time	0.07	0.20	2.46
New Page	0.06	0.34	4.30
Underline	0.06	0.11	1.40
Gold Key	0.05	0.02	0.25
< >	0.05	0.09	1.60
Halt	0.04	0.11	1.96
Cut	0.04	0.12	2.36
Tab Pos	0.04	0.13	2.55
Top of Document	0.04	0.53	11.45
Rub Line	0.03	0.03	0.80
Control characters	0.03	0.11	2.71
Lower Case	0.03	0.04	1.11
Swap	0.03	0.03	0.81
Sentence	0.02	0.06	1.89
Search	0.02	0.28	9.43
View	0.02	0.14	4.78
Gold Paste	0.02	0.05	1.69
Center	0.02	0.04	1.37
Paste	0.02	0.10	4.79
Get Document	0.01	0.17	9.56
Bottom of Document	0.01	0.40	22.95
Advance to End	0.01	0.12	7.74
Rub Sentence	0.01	0.02	1.11
Hyphen Push	0.01	0.01	1.08
Cont Search	0.01	0.02	2.13
Backup to End	0.01	0.13	17.78
Subscript	0.01	0.01	1.80
Paragraph Marker	0.00	0.01	1.62
Superscript	0.00	0.02	2.53
Command	0.00	0.00	3.75
Hyphen Pull	0.00	0.01	0.64
Bold	0.00	0.00	1.33
Remove Underline	0.00	0.01	0.48
Dead Key	0.00	0.02	3.34
User Defined Key	0.00	0.01	8.17
Page Marker	0.00	0.00	6.62
Cont Search & Sel	0.00	0.00	1.12
Gold Cut	0.00	0.00	1.16
Undelete Word	0.00	0.00	3.18
Replace	0.00	0.00	1.51
Library	0.00	0.00	2.34
Print Hyphen	0.00	0.00	0.00
Abbreviation	0.00	0.00	0.00
Remove Bold	0.00	0.00	0.00
Undelete Character	0.00	0.00	0.00
Totals	100.00	100.00	0.82

Table 3: WPS Keystroke Frequency Table

FROM | TO--> V	A/N	Left	Right	Delete	Down	Up	Return	Tab	Word	Ins/Ovr
A/N	.92	.00	.00	.03	.01	.00	.02	.00	.00	.00
Left	.03	.93	.01	.00	.01	.00	.00	.00	.00	.00
Right	.03	.01	.92	.01	.00	.00	.00	.00	.00	.01
Delete	.25	.00	.01	.70	.01	.00	.01	.01	.00	.00
Down	.02	.08	.03	.01	.78	.03	.01	.00	.00	.00
Up	.03	.04	.06	.00	.04	.78	.01	.00	.00	.00
Return	.38	.00	.02	.02	.04	.07	.19	.24	.00	.00
Tab	.52	.01	.01	.05	.03	.01	.00	.35	.00	.00
Word	.00	.02	.01	.00	.00	.00	.00	.00	.86	.00
Ins/Ovr	.42	.02	.06	.21	.06	.01	.02	.01	.00	.16

Table 4: EPT Keystroke Transition Probabilities (n = 1)

these statistics, we assigned the time preceding each keystroke to the keystroke in question. Analogous computations based on the time following each keystroke give very similar results.

Table 2 shows that typing accounted for about one half of editor usage; both in terms of keystroke frequency and time. The keystroke-level editing functions showed a non-uniform distribution of usage frequencies; the most frequently used functions are over 1000 times more frequent than the least frequently used functions. The top five editing functions account for more editing time than all the other functions combined. Four of these functions are the up, down, left, and right cursor movement functions — collective use of these functions accounted for about one quarter of user editing time. Single character deletion was the next most frequently used function, accounting for about 7% of users' time.

Table 3 shows summary statistics for the WPS work sample. The format of Table 3 is the same as that for Table 2. The table treats "gold" key editing functions as a single keystroke. Typing accounted for about one half of user time, as was the case with the EPT work sample. The distribution of times across functions is again highly non-uniform. The three most often used functions (Advance, Line, Backup) are all cursor movement functions, similar to the arrow function in EPT. These three functions account for just over 25% of total editing time. Single character deletion (Rub Character) was the next most frequent function.

3.3. Keystroke Transition Analysis

In addition to computing results at an individual keystroke level, we also examined transitions between keystrokes. Table 4 shows results of such analysis in the form of a keystroke transition matrix. The table shows keystroke transitions for the 10 most frequently used EPT functions. The function preceding the transition is shown along the left-hand margin. The function following the transition is shown along the top of the table. The entries in the matrix show the observed probability of transition from the row keystroke to the column keystroke. For example, the upper-left hand cell pertains to transitions between alphanumeric keys (printing characters). The transition probability of .92 indicates that if the user has just entered a printing character, the probability is .92 that the next keystroke will also be a printing character. The sum of all the probabilities in a row is always 1.0 (accounting for all transitions). Note that this is not true of the entries in Table 4; only the upper-left hand corner of the full 29 by 29 entry matrix is presented.

Table 5 shows a transition matrix similar to Table 4, except the upper entry in each cell shows the actual number of transitions of each type that took place. In addition to computing actual transitions, we also estimated how many transitions ought to occur, based on the assumption that the entries in the matrix are determined only by the marginal totals. The empirical matrices differ substantially from expectation, given such a hypothesis.

FROM TO--> / V	A/N	Left	Right	Delete	Down	Up	Return	Tab	Word	Ins/Ovr	Totals
	252543	745	1372	9072	2199	708	4487	1153	1	307	273504
A/N	146536	35279	34231	20100	15506	8628	3778	2774	922	915	
	2156	61052	648	298	485	310	150	105	3	304	65846
Left	35278	8494	8241	4839	3733	2077	909	668	222	220	
	1971	679	58923	774	350	254	190	32	10	402	63893
Right	34232	8242	7997	4696	3622	2016	882	648	216	214	
	9280	155	385	26173	536	136	288	214	1	153	37516
Delete	20100	4839	4695	2757	2127	1184	518	381	126	126	
	464	2347	965	311	22722	902	191	33	10	96	28936
Down	15503	3732	3622	2127	1641	913	400	293	98	97	
	420	659	1003	80	703	12637	104	59	10	62	16105
Up	8629	2077	2016	1184	913	508	222	163	54	64	
	2664	4	112	120	287	478	1371	1688	0	21	7045
Return	3775	909	882	518	399	222	97	71	24	24	
	2702	41	55	253	174	41	22	1830	0	8	5177
Tab	2774	668	648	380	294	163	72	53	17	17	
	4	28	23	5	2	6	3	0	1482	1	1720
Word	922	222	215	126	98	54	24	17	6	6	
	711	32	102	358	104	17	29	12	0	279	1707
Ins/Ovr	915	220	214	125	97	54	24	17	6	6	
Totals	273513	65850	63894	37518	28943	16105	7051	5178	1721	1707	510503

Key to entries:
observed number of transitions
expected number of transitions

Table 5: EPT Keystroke Transition Matrix, $n = 1$ (Totals from full matrix)

We were interested in describing the degree of organization, or degree of non-randomness, inherent in the users' keystroke patterns. Suppose, as an example, that keystroke sequences are entirely random. We would then expect the empirical transition frequencies to be very close to those that we would predict on the basis of the marginal frequencies. Clearly, this is not the case for the empirical data shown in Table 5. A comparison of obtained with expected frequencies shows the extent to which the actual keystroke behavior deviates from random expectation. We can compute a chi-square statistic using the expected and obtained frequencies. This shows the two matrices to be significantly different at the .001 level (chi square = 2,249,886; df = 273). Another way of expressing the same idea is with an index of predictive association [7, p. 747]. This is an index varying from 0.0 to 1.0, which shows the extent to which knowing the initial state improves the accuracy of prediction of a subsequent state. The index of predictive association for the data in Table 5 is 0.73 (with a 95% confidence limit of less than .01). For a transition matrix based on random key sequences, the index would approach 0.0.

We were also interested in attempting to describe the degree to which keystroke behavior is organized over longer sequences. Table 5 shows transitions between keystroke n and keystroke $n + 1$, for all n in our sample. We also computed transition matrices for transitions between keystrokes n and $n + m$ for m between 1 and 25. Table 6 shows such a matrix, in this case for $m = 12$. That is, the column keystrokes are those that follow the row keystrokes with a delay of 11 intervening strokes. The intent of this analysis is to see how far ahead in a sequence of strokes we can make good predictions, based on knowledge of the stroke at the start of the sequence.

We have generated expected numbers of keystrokes for the n to $n + 12$ transitions; these are also shown in Table 6. Notice that the actual frequencies still deviate substantially from the random predictions, but that the deviations are smaller than those in Table 5 (n to $n + 1$) predictions. This

FROM TO-->	A/N	Left	Right	Delete	Down	Up	Return	Tab	Word	Ins/Ovr	Totals
	211716	11597	7323	21920	4312	3086	4599	3394	87	306	270026
A/N	146238	35322	33721	20093	13774	8412	3668	2771	860	868	
	11385	37604	6316	1885	3518	1947	357	368	171	380	64958
Left	35179	8497	8112	4834	3314	2024	882	667	207	209	
	10686	6115	36242	2864	3118	1568	404	221	106	374	62708
Right	33961	8203	7831	4666	3199	1954	852	644	200	202	
	22174	1520	2090	8354	972	508	452	425	19	79	36918
Delete	19994	4829	4610	2747	1883	1150	502	379	118	119	
	3423	4920	5133	684	8500	3198	257	125	109	129	27460
Down	14872	3592	3429	2043	1401	855	373	282	87	88	
	1980	2470	3123	318	2960	3866	125	77	110	94	15657
Up	8479	2048	1955	1165	799	488	213	161	60	50	
	4613	112	479	433	188	344	379	97	14	12	6742
Return	3651	882	842	502	344	210	92	69	21	22	
	3506	264	172	487	64	81	96	370	1	13	5102
Tab	2763	667	637	380	260	159	69	52	16	16	
	74	139	104	22	149	96	10	4	702	7	1638
Word	887	214	205	122	84	61	22	17	5	5	
	455	239	339	103	176	83	9	12	2	158	1622
Ins/Ovr	878	212	203	121	83	61	22	17	5	5	
Totals	271408	65556	62584	37292	25564	15612	6808	5143	1596	1611	501160

Key to entries:
observed number of transitions
expected number of transitions

Table 6: EPT Keystroke Transition Matrix, $n = 12$ (Totals from full matrix)

result is also expressed in the value of the index of predictive association which is 0.30 for the data in Table 6.

Figure 1 plots the index of predictive association as a function of the number of transitions between keystroke n and $n + m$ for $m = 1$ to 25. For EPT this function is plotted by circles. Notice the smooth decay of the the value of the function, indicating that it becomes progressively harder to predict the longer range one's prediction is.

The transition analysis described above was also performed for the WPS work sample. Table 7 shows the transition probabilities for the 10 most frequently used functions. The overall picture of editing that emerges from these data is very similar to that for EPT; the most frequent transitions are generally along the diagonal. On Figure 1 the predictive association for the WPS sample is depicted as triangles. The general shape of the curve is very close to that for the EPT work sample; it starts at a slightly lower level, crosses over, and appears to level off at a somewhat higher level, but the overall shape of the curve is almost identical.

3.4. State Analysis

In addition to analyzing our data at the level of individual keystrokes, we can define sequences of keystrokes as editing states and analyze the data at that level. This gives a more global picture of user behavior. A state consists of a sequence of keystrokes relating to only one editing function. For example, Type state is defined as an uninterrupted string of text. Cursor state is defined as any uninterrupted string of cursor positioning operations. Erase state involves any string of keystrokes concerned with deleting text. Edit state is eclectic but includes any continuous string of keystrokes concerned with altering or moving text. Start state is automatically entered when the editor is turned on; Finish is always the last state in an editing session. Pause state is entered after 150 seconds of user inactivity and ends with the next keystroke. Finally, Help state involves asking for on-line documentation. By definition no state may follow itself and no state may be nested within another state.

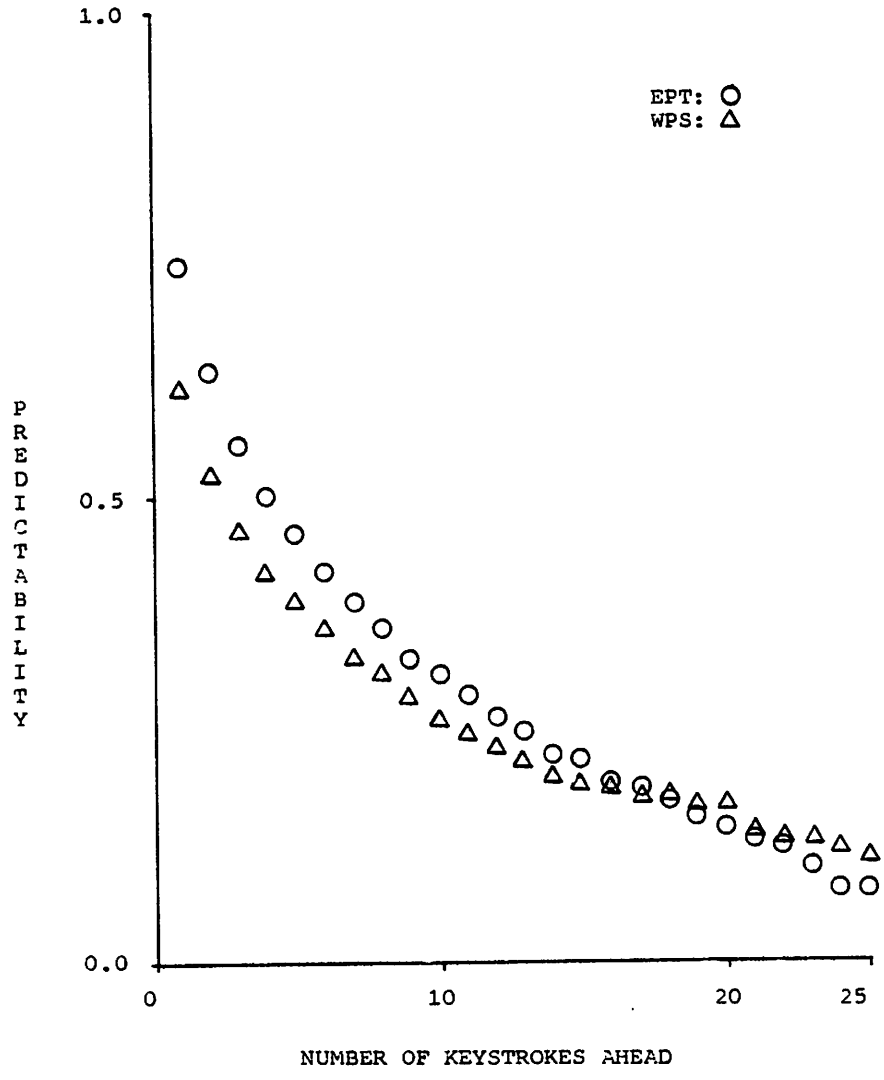


Figure 1: Predictability of Future Keystrokes as a Function of Lookahead Length

Table 8 shows the state analysis for EPT, including the number of occurrences of each state, the percent of keystrokes in each state and the mean number of keystrokes per state. The 739 Start states mean that there were 739 editing sessions in the work sample. On average, users in Type state tended to generate strings of 16.5 characters (this average is somewhat misleading, the frequency distribution is highly skewed with a mode of 1 character and a very long tail — the longest recorded string was 512 characters). Erase state consisted overwhelmingly of single-character-delete strokes so the 3.6 keystroke average is very close to the mean number of characters deleted per state.

We are not able to present an identical analysis for WPS as of this writing because it has a much more complex command syntax. However, the mean length of uninterrupted character strings was 17.2, very close to the 16.5 figure for the EPT sample. Table 9 shows the percentage of keystrokes for the WPS sample falling into some of the states defined for EPT. Even this incomplete comparison reveals a striking similarity of usage patterns.

3.5. Transitions Between States

We earlier discussed transitions between keystrokes. A similar analysis is possible at the state level. Table 10 shows a transition matrix for the EPT states. This matrix shows transitions between state n and state $n + 1$ for all states in the work sample. Notice that all the diagonal entries are zero; this follows from the definitions of the states themselves. For each cell in the matrix there are three entries. The first shows the row transition probability. The

FROM TO--> | V	A/N	Advance	Line	Backup	Rub Chr	Return	Tab	Word	Del Chr	Para
A/N	.92	.02	.00	.01	.02	.02	.01	.03	.00	.00
Advance	.03	.78	.10	.03	.01	.00	.00	.02	.00	.00
Line	.01	.09	.76	.08	.01	.01	.01	.00	.00	.00
Backup	.06	.03	.12	.68	.02	.01	.01	.03	.01	.00
Rub Chr	.30	.07	.00	.02	.57	.01	.01	.00	.00	.00
Return	.29	.08	.01	.03	.01	.38	.09	.00	.00	.00
Tab	.49	.09	.01	.03	.05	.01	.29	.00	.01	.00
Word	.08	.08	.00	.06	.08	.00	.02	.58	.01	.00
Del Chr	.07	.12	.01	.03	.01	.00	.01	.00	.74	.00
Para	.00	.08	.03	.07	.00	.00	.00	.00	.00	.73

Table 7: WPS Keystroke Transition Probabilities (n = 1)

State	Number of states	Percent of keystrokes	Mean keystrokes per state
Type	17298	56.97	16.63
Cursor	9446	34.39	18.60
Erase	11729	8.27	3.60
Edit	253	1.20	24.12
Finish	739	0.14	1.00
Help	66	0.02	2.27
Start	739	0.00	0.01
Pause	204	0.00	0.00

Table 8: EPT State Summary Table

State	Percent of total keystrokes
Type	59.16
Cursor	34.45
Erase	4.44
Other (includes Edit)	1.95

Table 9: WPS State Summary Table

second entry shows the observed number of transitions and the third entry shows the number of transitions to be expected on the basis of the marginal totals (random expectation). Thus, from Type state there is a .57 probability of entering Erase state next and a .41 probability of entering Cursor state. For the most part, there is fairly close agreement between the obtained and expected transition frequencies.

From the transition frequencies we can compute the index of predictive association, as we did for the keystroke transitions. Further, we can compute this index for transitions n to $n + m$, as before. Figure 2 shows the index for up to 25 states ahead. The index falls off slowly as a function of number of states ahead and the values appear to form an alternating series with even-numbered lookaheads consistently higher than adjacent odd-numbered lookaheads. Table 11 helps explain this pattern. It shows a transition matrix for transitions between states n and $n + 2$. Notice the high transition probabilities along the diagonal and the corresponding high discrepancy between the observed and expected values along the diagonal. This same pattern is evident in the other even-numbered transition matrices. It means that when users leave a state, they tend to return directly to it after they have entered a single intervening state.

FROM TO-->	Type	Erase	Cursor	Start	Finish	Edit	Pause	Help	Totals
	0	.57	.41	.00	.02	.00	.00	.00	
Type	0	9800	7032	0	336	13	89	22	17292
	0	9355	7129	0	476	163	132	36	
	.86	.00	.13	.00	.01	.00	.00	.00	
Erase	10056	0	1551	0	87	9	20	6	11729
	9355	0	2133	0	142	49	39	11	
	.74	.20	.00	.00	.02	.02	.01	.00	
Cursor	7024	1847	0	0	239	224	95	17	9446
	7130	2133	0	0	108	37	30	8	
	.15	.06	.73	.00	.04	.01	.00	.01	
Start	112	46	537	0	31	6	0	7	739
	476	142	108	0	7	2	2	1	
	.00	.00	.00	.00	.00	.00	.00	.00	
Finish	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	
	.11	.06	.80	.00	.02	.00	.00	.01	
Edit	28	15	203	0	5	0	0	2	253
	163	49	37	0	2	0	1	0	
	.31	.09	.41	.00	.19	.00	.00	.01	
Pause	63	18	83	0	38	0	0	2	204
	132	39	30	0	2	1	0	0	
	.16	.05	.71	.00	.05	.02	.00	.00	
Help	9	3	40	0	3	1	0	0	56
	36	11	8	0	1	0	0	0	
Totals	17292	11729	9446	0	739	253	204	56	39719

Key to entries:
transition probabilities
observed number of transitions
expected number of transitions

Table 10: EPT Editing State Transition Matrix ($n = 1$)

4. Discussion

4.1. General Conclusions

A number of general conclusions about screen-oriented text editing are suggested by the results. The distribution of use across editing functions is grossly non-uniform. Certain functions are used constantly, others almost never. A rough rule of thumb that emerges from our data is that free usage text editing consists of about 1/2 typing, about 1/4 cursor movement, about 1/8 deletion, and 1/8 all other functions put together. This rule has been shown to apply in two different situations: knowledge workers creating documents with an experimental text editor, and secretaries transcribing and updating documents on a commercial word processing system. Considering the differences in the editors, the workers, and the tasks, the correspondence between the work samples is striking. It implies that it may be possible to optimize a single editor for a variety of users.

Another finding to emerge from the state analysis is that users tend to alternate editing subtasks; they tend to return to the state they left one state before. For example, if a user is typing, he tends to leave that state to do one other thing (perhaps deleting) and then immediately returns to typing.

4.2. Extending the State Analysis

The set of states presented here was sufficient for analyzing EPT, but not for analyzing more powerful editors. Two new states, Format and Manage, can handle these more sophisticated functions. Format includes formatting commands that alter the outward appearance of a document. Manage includes commands for managing multiple files and windows while within an editor.

Since EPT was a simple editor, it was easy to build the necessary state analysis programs. State analysis is much more difficult than keystroke analysis for any large production editor due to the extensive parsing that must be done. This problem is alleviated in situations where the analysis can be done using the code from the editor's parser.

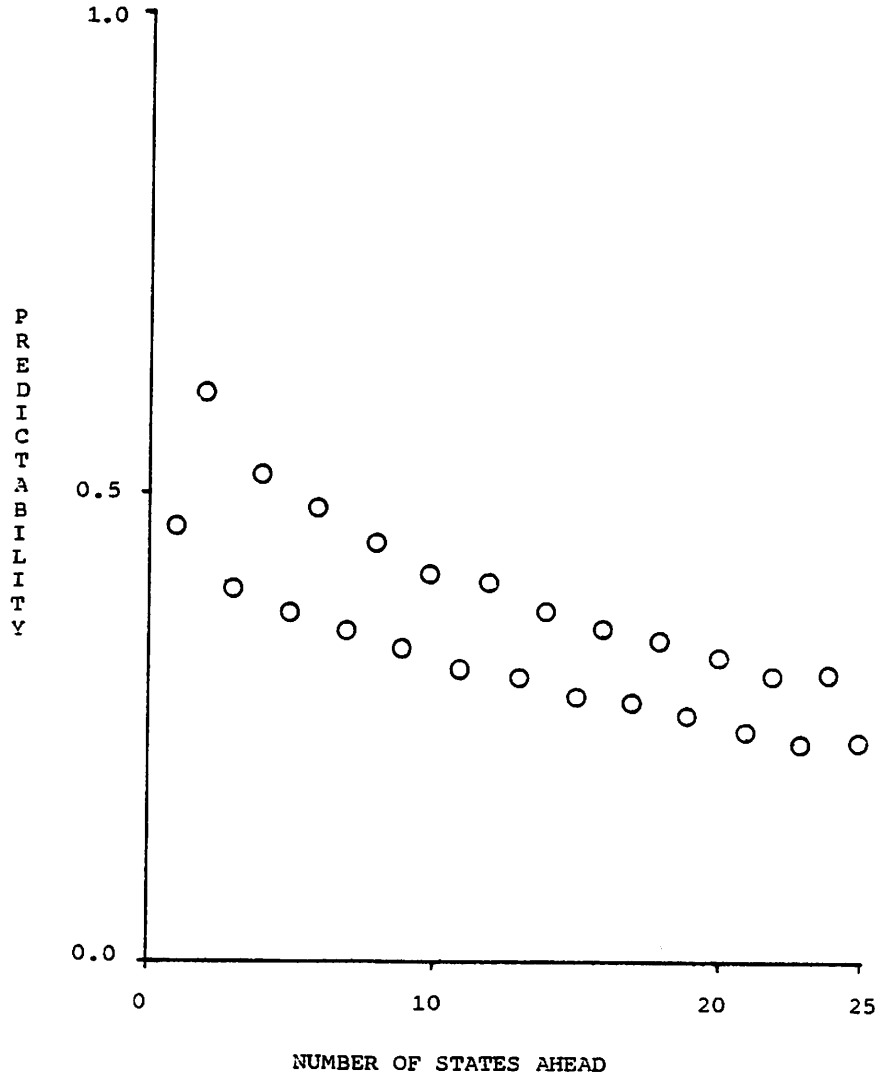


Figure 2: Predictability of Future States as a Function of Lookahead Length

4.3. Relation to Past Research

The analysis gives us a picture of editing based on actual usage data, rather than on some abstract model. As such, the data presented here provide a good test of the relevance of these abstract models and can be seen as a complement to them.

4.4. Implications for Editor Design

Data such as these can be used to drive the software architecture of text editors. Raw frequency-of-use data indicates which aspects of editor design should be optimized. For instance, cursor placement is the single most time consuming editing function next to typing itself. Thus, the mechanics of cursor positioning deserve considerable engineering attention and optimization. In addition, the transition data might be used to tune editor software to the behavior patterns of users. For example, faster performance could be obtained by reordering the alternatives within a parser, and by improving the locality of reference within the editor.

4.5. Implications for Keyboard Design

Beyond its implications for editor design, this data provides a sound empirical basis for the application of McCormick's [8] principles of keyboard design. In order to place keys in terms of either their overall usage frequency or the frequency of transitions between keystrokes one must have data of the type provided here.

FROM | V / TO-->	Type	Erase	Cursor	Start	Finish	Edit	Pause	Help	Totals
	.88	.07	.03	.00	.01	.01	.00	.00	
Type	14936	1117	514	0	214	100	61	14	16956
	7473	5082	3875	0	308	107	89	21	
	.07	.73	.17	.00	.02	.00	.00	.00	
Erase	837	8539	1934	0	219	38	62	13	11642
	5131	3489	2661	0	211	74	61	15	
	.09	.19	.69	.00	.02	.00	.00	.00	
Cursor	788	790	6329	0	229	11	42	18	9207
	4058	2760	2104	0	167	58	48	12	
	.62	.19	.10	.00	.04	.02	.03	.00	
Start	440	137	68	0	28	13	21	1	708
	312	212	162	0	13	4	4	1	
	.00	.00	.00	.00	.00	.00	.00	.00	
Finish	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	
	.37	.14	.13	.00	.02	.32	.01	.01	
Edit	91	36	32	0	4	80	3	2	248
	109	74	57	0	5	2	1	0	
	.40	.28	.16	.00	.08	.00	.08	.00	
Pause	67	46	26	0	14	0	13	0	166
	73	50	38	0	3	1	1	0	
	.40	.34	.11	.00	.00	.09	.04	.02	
Help	21	18	6	0	0	5	2	1	53
	23	16	12	0	1	0	0	0	
Totals	17180	11683	8909	0	708	247	204	49	38980

Key to entries:
transition probabilities
observed number of transitions
expected number of transitions

Table 11: EPT Editing State Transition Matrix ($n = 2$)

4.6. Implications for Benchmark Tests

Benchmarking is an appropriate testing method for text editors when there is an interest in making comparisons among existing editors, or improving the performance of an editor which may be under development. The best way one can develop a realistic set of benchmark tasks for a text editor is on the basis of actual usage statistics. Roberts has made the first attempt at developing such benchmark tasks. These tasks include a test of speed of use for experts, developed on the basis of an informal study.

In order to evaluate the representativeness of Roberts' benchmarks two EPT experts performed Roberts' expert tasks. Their data was then compared to real EPT behavior in Table 12. Table 12 shows that users spent much more time in the Edit and Cursor states when performing Roberts' tasks than they did in free use.

State	Actual Usage	Roberts' Task
Type	53%	14%
Cursor	32%	49%
Erase	10%	11%
Finish	3%	1%
Edit	2%	25%
Help	0%	0%
Start	0%	0%

Table 12: EPT State Usage as Percentage of Working Time

An increased knowledge of actual user behavior should enable researchers to develop benchmarks which more closely reflect actual behavior. In addition, given the similarities in user behavior, newly developed benchmarks should be widely applicable.

References

1. Card, Stuart K., Moran, Thomas P., and Newell, Allen. Computer text-editing: an information processing analysis of a routine cognitive skill. *Cognitive Psychology 12* (1980), 32–74.

2. Card, Stuart K., Moran, Thomas P., and Newell, Allen. The keystroke-level model for user performance with interactive systems. *Comm. ACM 23* (1980), 396–410.

3. Embley, David W. and Nagy, George. Empirical and formal models for the study of computer editors. In *Computing Skills and the User Interface*, M. Coombs and J. L. Alty, Eds., Academic Press, London, 1981.

4. Embley, David W. and Nagy, George. Behavioral aspects of text editors. *Comput. Surv. 13* (1981), 33–70.

5. Hammer, John M. The human as a constrained optimal text editor. Ph.D. Th., University of Illinois, June, 1981.

6. Hammer, John M. and Rouse, W. B. Analysis and modeling of freeform text editing behavior. Proc. 1979 Internat. Conf. Cybernetics and Society, Oct., 1979, pp. 659–663.

7. Hays, William L. *Statistics for the Social Sciences.* Holt Rinehart Winston. New York, 1973. Second Edition.

8. McCormick, Ernest J. *Human Factors in Engineering and Design.* McGraw-Hill, New York, 1976. Fourth Edition.

9. Roberts, Teresa L. Evaluation of computer text editors. Report SSL-79-9, Xerox PARC, Nov., 1979. Stanford Ph.D. dissertation.

VIRTUAL EDITING:
II. The User Interface

H. A. Wilder

N. F. Maxemchuk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the second half of a two-part report on Virtual Editing. In the first paper, virtual editing was defined and its implementation described. In this paper, a user interface to the editor is described and some of the problems of introducing a new form of communication which conceptually merges various media are discussed. The aim is acceptance by the office worker, or eventual end user. Possible applications for this technology are suggested. In addition, the use of a mouse in the system as a way of entering the user commands is explained.

I. INTRODUCTION

As the automated office of the future becomes more and more a reality of today, the computer and its associated technologies are assuming responsibilities for tasks formerly performed by the office workers. Most of these tasks revolve around the communication of some type of information, whether it be by voice, written text, graphic pictures or some other medium. Some specific services involving these modalities would be a dictation service, with text typed from recorded voice, or a messaging service which supports voice, text or graphics[1].

The computer has been found to be a useful tool for the storage of information and much work has been done on the ways in which that information is composed, retrieved and updated. Up until just recently, work done on the composition, retrieval, and editing of documents represented in the various media (text, voice[2], and graphics), was focused more on the particular medium involved, rather than the information contained.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0041 $00.75

Compatibility between the editors was therefore not an issue, and since a keyboard and CRT terminal were most readily available, text editor interfaces became the most prevalent. This is not to say that text was assumed to be the best medium for all the information, it was just the most accessible.

The technology is now present which allows the type of medium used be determined by the person or application, not by the availability of the technology. Much attention is now being paid to the ways in which documents represented in differing media can be merged and manipulated, and new approaches and implementations have been suggested[3], as well as quite a few ideas for application of this technology.

In "Virtual Editing: I. The Concept"[4], we define virtual editing as those editing functions which are common to any type of medium editor, and therefore independent of the medium type. Virtual editing involves changing, moving, deleting part of the document, or inserting new information in a document. The more medium specific editing functions (e.g. changing the endpoints of a line in graphics) are not included in the virtual editor. A document can be thought of as an ordered series of one or more segments of stored information. Each segment can be in the medium which is best suited to represent and communicate that information.

Conceptually, editing is a re-ordering of new or existing segments, which may be divided into smaller segments. A segment can be thought of as a block of information addressed by beginning and end pointers to its location within a document. These pointers can be reset so that the segment (and the document that the segment is a part of) can be split apart and re-assembled in a new way. The operation of re-ordering segments need not be concerned with the type of medium of which the segment is composed. For example, given a document composed of a text segment followed by a voice segment, the middle can be deleted by dividing the text segment at the desired spot, dividing the voice segment at the desired spot and then joining the first half of the text to the second half of the voice.

In the prototype of the editor that has been built, a document is 'played back' or 'displayed' for the user, who has the option of going forward or backward in the document.

In addition, a document can be skimmed quickly or studied more slowly. Drivers for each type of medium could easily be included in the editor, although to date, only text and voice drivers have been written. Since part of a previous project[5][6] was the implementation of a library of routines to interface with a speech storage and editing facility, the problem of writing a voice driver was reduced to a straight-forward inclusion of the appropriate routines.

Due to the novelty of multi-media editing, the problem of presenting it to a user in an understandable manner becomes crucial to its acceptance. This paper is a first attempt at a solution.

II. USER INTERFACE

As shown in "Virtual Editing: I."[4], editing can be mapped into layers, with the drivers for the medium dependent operations at the top and the medium independent operations on a lower layer. This layering of functions is similar to the International Organization for Standardization (ISO) protocol layers for open system interconnection. In keeping with this concept, the designs of both the editor and the user interface are divided into logical 'layers' of functionality. This not only provides a way of keeping user control and application distinct from the specific medium drivers as far as the computer is concerned but also allows the user a view of the document from a level that gives transparency to the medium type.

In our editor, the application program uses menu selection for user control, totally neutral to the type of medium in the document. At present this is loosely applied to a messaging service. For the actual review, creation, or re-positioning of segment boundaries within a specific segment, the appropriate medium driver is called. An effort was made to be as consistent as possible with each driver from the user's view. In addition, to further this clarification of control and medium, a way for the user to issue any command (e.g. moving the cursor, increasing the speed of a voice playback, selecting the desired editing function) is provided, which is wholly independent from the entering of the information. For example, to make a document composed entirely of speech, a telephone is needed to enter the voice, but it is not necessary to touch the keyboard which is reserved exclusively for the entering of text.

As suggested by Limb [7], a mouse [8] was used to achieve the objectives of keeping the entering of control and data separated, and at the same time using the same apparatus in all media. The mouse has been used frequently to position the cursor on the screen by rolling the mouse, and to select a screen position or mode by using the buttons. A good example of this is in Xerox's text-processing system *Bravo* [9]. In our system, movement of the mouse allows one to jump to the desired item in a menu, and select it using a button on the mouse. The cursor is only allowed to land on one of the twelve menu items (the menu consists of two columns of up to six possible choices each). This guarantees that the user will select an appropriate item with minimal exactness required.

Direction and speed of the display are also controlled by the mouse. In text, movement of the mouse was mapped onto movement of the cursor. Selecting a position in the text is done by activating the buttons when the cursor is at the desired spot. Long text can be scrolled forward or backward on the screen with upward or downward movements of the mouse. In like manner, jumping forward or backward in voice is done with a corresponding forward or backward movement of the mouse. The playback speed can be increased or decreased by appropriate mouse movement in the same way that the speed of scrolling through text can be controlled by mouse movement. A quick movement downward will quickly 'skim' the user through a text segment; the same movement will increase the playback speed of voice, effecting a 'quick listen' through it. Moves forward and backward in voice are done along a 'time line', complete with 'bouncing-ball', showing the relative position in the voice at that moment (FIG 1). The ball normally bounces forward as the speech is played. When the user jumps forward or backward in the voice document, a corresponding movement of the ball is done on the time line. This graphic representation of the voice document provides a means of 'seeing' where one is in a temporal domain.

At present, alternate displays showing position in the entire document are being investigated, aiming for a method of graphically showing composition of the document as well. A position in the voice document is selected by the same select button as in text, when the desired spot is heard by the user. Since text is a static medium, it is relatively easy for the user to take time and think about positioning of the marker, as well as verifying its ultimate position by sight. In voice, selecting a position is done as the voice is playing back; the user cannot see where it will be until the second he/she hears the desired word. Because of this relative lack of control in marking a particular spot in voice as opposed to text, a two-step procedure is used. The first step is to select the approximate position; most times this will be where it was intended. The second step includes verification of the position and possible re-positioning. The document is played back from the marker; if it is not in the desired position, forward or backward movement of the mouse will move the marker accordingly, the position being verified each time it is moved (FIG 2). This seems to provide the same amount of precision for editing in voice as in text.

This points up some basic similarities and differences between voice and text. Voice is an 'active' medium; it moves itself forward, stopping automatically when the document is done. In text there is no way of knowing when the user has completed reading. It is a 'passive' medium which requires some action on the part of the user to move forward, and finish. To overcome this disparity, the user now has to 'quit' each segment, regardless of the medium. A voice segment will still stop when playback is complete, but the user must activate continuation to the next segment. This also

prevents being automatically thrown into the next segment before the user is ready, giving as much time as necessary to review each segment.

Another difference involves the representation of the medium in some symbolic form. Voice as mentioned before can be shown in a kind of time-line since it is a linear mapping from time to a directed line. Text, on the other hand, may be measured in units of words or sentences but the translation of this into a line showing position has not yet been resolved in a manner that corresponds with voice. As of now, another difference exists in the state of the technology used to perform pattern recognition in the different types of media. The area of speech recognition is still being explored, so certain editing functions which are common to most text editors (e.g. search for ASCII string which is relatively easily implemented in text) were omitted since an effort was made to have consistent operations throughout the document.

III. THE MOUSE

The mouse* is rolled along a surface by the user. It transmits the horizontal and vertical directions of these movements, as well as the status of the three buttons located at the front of the mouse. Transmission of information from the mouse is done through specially constructed hardware which interfaces it directly to the host computer (a VAX 11/780) (see Appendix).

It was decided that the first and third buttons on the mouse should always correspond to the same action, no matter what mode the user is in. The first button will always SELECT. In a menu, this implies choosing the item which the cursor is presently positioned on. In text or voice this is used for editing purposes, to select the desired position at which an edit should take place. The middle is used for PAUSE/RESUME when recording or playing back speech. If the user has paused while playing back a voice segment, any type of movement of the mouse (increase/decrease speed, forward/backward jump) will be taken as a request to resume. The parallel operation to PAUSE/RESUME in text, pausing and resuming during automatic scrolling of a text segment, has not been implemented as of yet. The third button will always QUIT. In a menu, this will take the user back to the previous menu. In reviewing text or voice, it will take the user to the next or previous segment depending on the direction the user has indicated when moving the mouse. If the user is not at the end or beginning of the present segment (attempting to go forward or backward), it is assumed that quitting review of the document is the desired action.

In a single transmission, sent every tenth of a second, the amount of movement in each direction is represented by a number ranging in value from 0 to 128. Although the mapping of the mouse movement to the cursor movement on the screen is seemingly linear; very short, slow movements of the mouse will give a very small movement of the cursor on the terminal screen while long, quick movements will give fast 'jumps' of the cursor. It was deemed desirable to give the necessary precision needed for exact placement, especially when editing, and also provide a fast method for scanning, especially in a long piece of text (more than 24 lines). A logarithmic mapping between modalities of various sensations has been found to be a very natural way of presenting cross-modalities for people[10]. To obtain this function, the range of mouse movements was divided into quartiles (FIG 3). In each quartile, the slope is doubled what it was in the previous quartile, resulting in a piecewise linear approximation to the exponential function

$$y=ce^{bx} \qquad \text{where} \quad c=\frac{e^{-\ln 2}}{2\ln 2} \text{ and } b=32\ln 2$$

This gives the desired amount of cursor movement on the screen when moving through characters in text; the maximum value transmitted by one long movement of the mouse (128) moves the cursor approximately halfway across a standard terminal screen. This can be easily modified by changing the value of c. The value c is increased slightly when mapping mouse movement onto number of seconds jumped in speech, since one second in voice doesn't mean as much as one character in text. The value c is decreased when moving between menu items, increasing or decreasing the speed of playback, and re-positioning the marker in voice (due to the need for added precision).

Diagonal movements posed somewhat of a problem. While it would be useful to be able to move in a diagonal, a straight horizontal or vertical movement should be made without possible drift of the user's hand taken as intentional. Informal testing showed that 45 degree diagonal movements were reasonable, with threshold levels set at 22 degrees (FIG 4). Anything below the threshold was discarded as drift, anything above was taken as an intended movement in that direction.

IV. CONCLUSION

It should be stressed that this is only a first attempt at introducing this concept. Refinements will come as the editor is used by different people. Fortunately, most of those who have seen the editor have come up with quite a few novel, on-the-spot, ideas and suggestions. Most of these revolved around the aforementioned problem of representing the document as a logical whole while showing the different types of media that compose the document so that the user can select to review the whole document or just parts. A way to represent text in a manner consistent with voice is needed so that a user gets an idea of the proportion of text and voice (and graphics, when that driver is built) in a single document.

* Manufactured by J.D. Nicaud,
Ecole de Polytechnique de Lausanne,
Lausanne, Switzerland

Questions have also been raised as to how this proportion affects the perception of the document. In a document consisting largely of text with short voice comments, the user may want to skip the voice 'annotations', and concentrate on the more important text information. In like manner the user may just want to hear the voice without the interruption of reading the small text notes scattered throughout. On the other hand, all the segments could be of equal importance, carrying information that should be considered as a whole.

Many of the answers will depend on the type of application for which the editor is used. Consider a dictation service: the speaker would start by recording a voice document. The typist who transcribes the document would gradually change that voice into typed text, returning the text product for approval. The speaker could re-submit the job with additional voice comments that the typist would listen to and then revise the text as specified. In a messaging service a text message may be forwarded on with voice comments, maybe highlighting, or refuting certain points in the message. A voice message may be sent with some information in text, possibly to emphasize the importance of that information, especially if a hard copy version should be saved.

The future of this technology will depend on its usefulness and usability. Some human factors testing has been conducted on the stored speech editor [11], but this only as a single medium editor. Despite this newfound capability for communicating information as a multi-media document, further testing and usage will be needed to determine its acceptance and applicability in the automated office of tomorrow.

V. ACKNOWLEDGEMENTS

The authors wish to thank Bob Allen, Mark Scerbo, Bob Rosin, and John Limb for their suggestions, help, and cooperation.

APPENDIX

Both the mouse and terminal input are transmitted through the special purpose hardware unit (FIG 5). Input from the terminal is transmitted normally to the host computer as long as there is no input from the mouse. Input from the mouse is transmitted to the host computer as three 8-bit characters (change in x direction, change in y direction and whether any of the three buttons had been pushed). It is prepended with the ASCII 'DEL' character to distinquish it from the terminal input. While the mouse is moving, it will continue to transmit the four 8-bit characters every tenth of a second. At this time transmission from the terminal is interrupted, and the data from the terminal will be discarded. It is assumed that the user will not access the mouse and the keyboard at the same time. All output from the host computer is sent directly to the terminal.

REFERENCES

1. Rosin, R. F. "Communication System Architecture", Proc. NTC'80, Houston, Tex., Nov. 1980, pp. 16.5.1-16.5.5
2. Maxemchuk, N. F., Wilder, H. A., Allen, R. B. "Experiments in Editing Speech", Proc. NEC'80, Chicago, Ill., Oct. 1980, pp. 228-232
3. Naffah, N. "Editing Multitype Document", Proc. International Workshop on Office Information Systems, Saint Maximin, France, Oct. 1981
4. Maxemchuk, N. F., Wilder, H. A. "Virtual Editing: I. The Concept", Proc. International Workshop on Office Information Systems, Saint Maximin, France, Oct. 1981
5. Maxemchuk, N.F., Wilder, H.A. "Experiments in Merging Text and Stored Speech", Proc. NTC'80, Houston, Tex., Nov. 1980, pp. 16.1.1-16.1.6
6. Maxemchuk, N. F. "An Experimental Speech Storage and Editing Facility", B.S.T.J. vol. 59,no. 8, Oct. 1980, pp. 1383-1395
7. Limb, J. O."Integration of Media for Office Services", Office Automation Conference Digest, 1981, Houston, Tex., Mar. 1981, pp. 353-355
8. English, W.K., Engelbart, D.C., Berman, M.L. "Display-Selection Techniques for Text Manipulation",IEEE Transactions on Human Factors in Electronics, vol. HFE-8,No.1, March, 1967
9. Lampson, B. W.,"BRAVO", Alto User's Handbook ,Xerox Palo Alto Research Center, Palo Alto, Calif., 1978, pp. 31-61
10. Brown, R.,Hernstein, R. Psychology ,Little, Brown and Company, Boston, 1975, pp. 403-409
11. Allen, R. B. "Composition and Editing of Speech", submitted to B.S.T.J., 1981

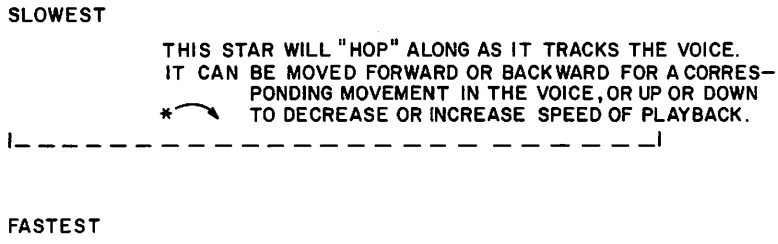

SLOWEST
THIS STAR WILL "HOP" ALONG AS IT TRACKS THE VOICE.
IT CAN BE MOVED FORWARD OR BACKWARD FOR A CORRES-
PONDING MOVEMENT IN THE VOICE, OR UP OR DOWN
TO DECREASE OR INCREASE SPEED OF PLAYBACK.
FASTEST

FIG 1 Screen displayed during the playback of a voice segment

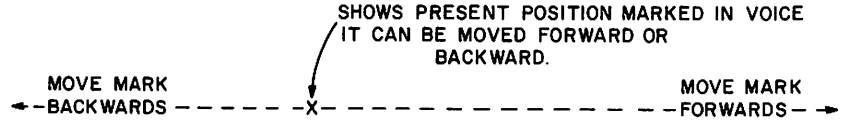


FIG 2 Screen displayed while user verifies or re-positions a marker set in the voice segment

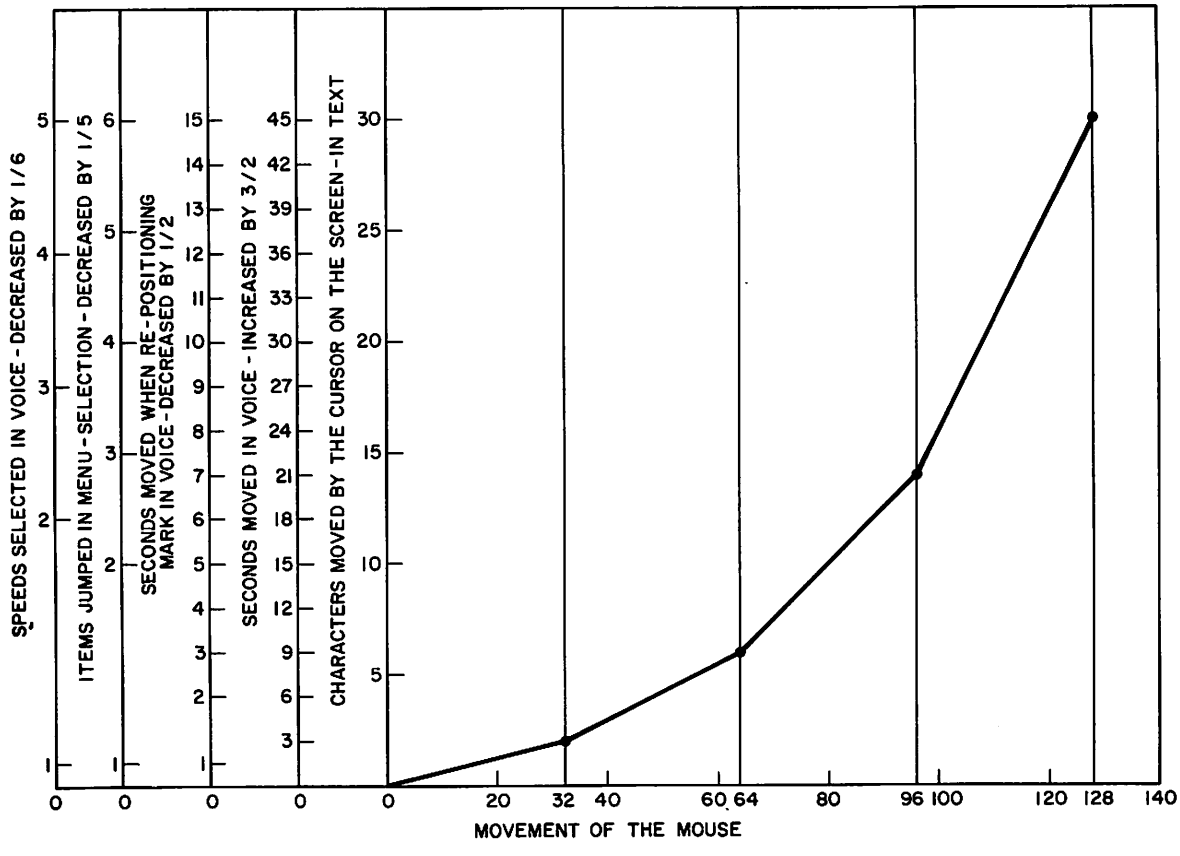

SPEEDS SELECTED IN VOICE - DECREASED BY 1/6
ITEMS JUMPED IN MENU - SELECTION - DECREASED BY 1/5
SECONDS MOVED WHEN RE-POSITIONING MARK IN VOICE - DECREASED BY 1/2
SECONDS MOVED IN VOICE - INCREASED BY 3/2
CHARACTERS MOVED BY THE CURSOR ON THE SCREEN - IN TEXT
MOVEMENT OF THE MOUSE
0 20 32 40 60 64 80 96 100 120 128 140

FIG 3 Functions used to map mouse movements to the various movements on the screen or in voice segments

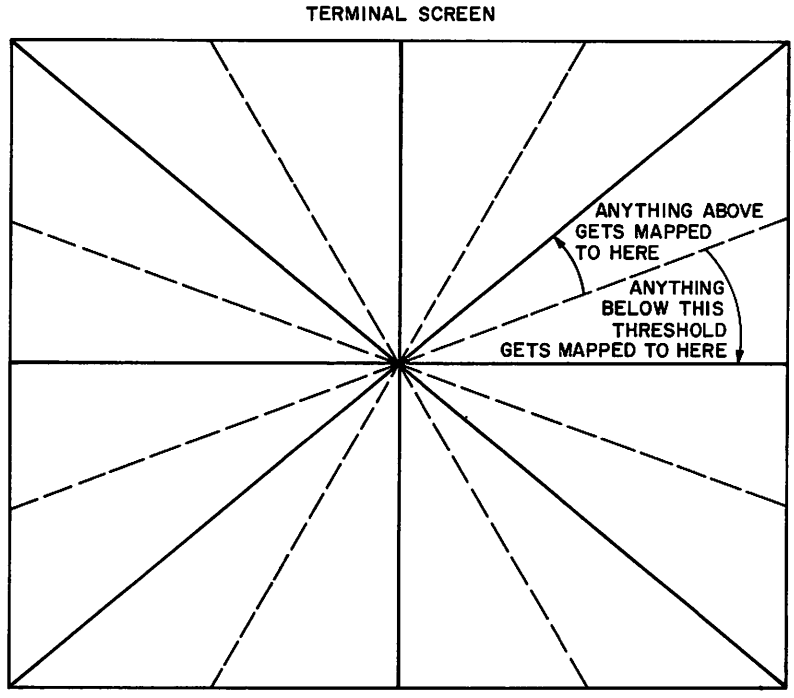


FIG 4 Thresholds for cursor movement to eliminate unintentional drifting of the user's hand

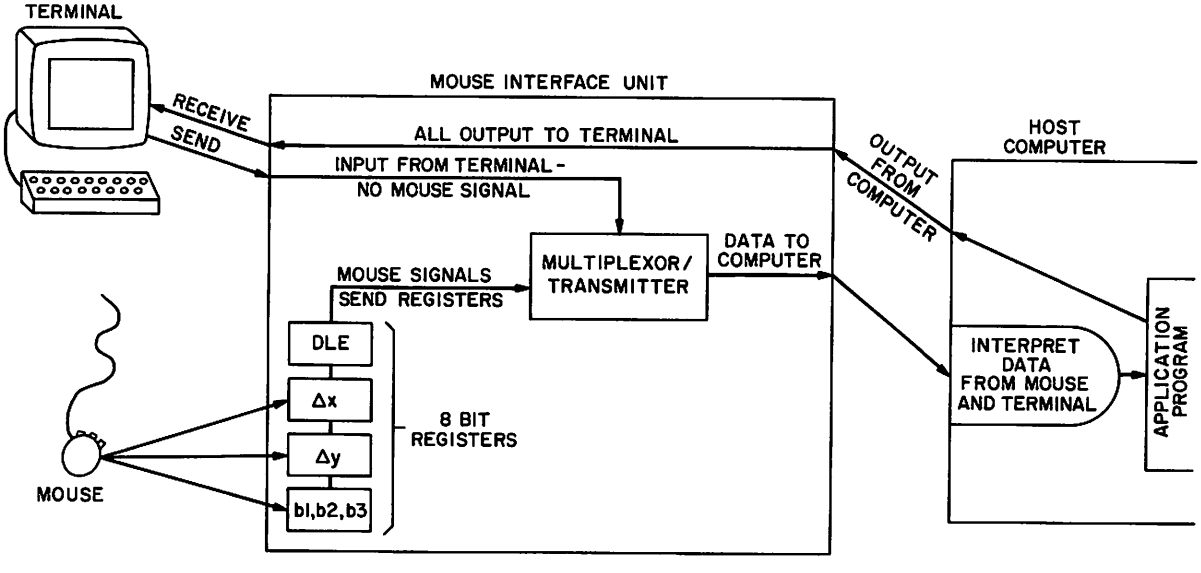


FIG 5 Block diagram of the hardware used to receive and transmit data from the mouse and the terminal. DEL precedes data from the mouse. Δx, Δy, b1, b2, and b3 indicate the x and y movement and the button status.

How Do People Organize Their Desks?

Implications for the Design of Office Information Systems

Thomas W. Malone

Cognitive and Instructional Sciences Group
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Extended Abstract

(Complete paper is in the supplement to these proceedings)

In this paper, I describe a series of case studies of how professional and clerical office workers organize the information in their desks and offices. Then I discuss a number of implications of these results for designing natural and convenient office information systems.

Two principal claims are made: (1) A very important function of desk organization is to *remind* the user of things to do, not just to help the user *find* desired information. Failing to support this function can seriously impair the usefulness of electronic office systems, and explicitly facilitating it can provide an important advantage for automated office systems over their non-automated predecessors. (2) The cognitive difficulty of categorizing information is an important factor in explaining how people organize their desks. Computer-based systems can help with this difficulty by: (a) doing as much *automatic classification* as possible (e.g., based on access dates), and by (b) including untitled "piles" of information arranged by physical location as well as explicitly titled and logically arranged "files."

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0047 $00.75

Interviews

The ten people interviewed were chosen to represent a broad spectrum of people who do much of their work at desks. Each person was interviewed for approximately one hour. The respondents were asked to give the interviewer "a tour of their office" explaining what information was where and why it was there. Some of the interviewees were also asked to find several documents in their office. Finally, all the respondents were asked a standard set of questions about how well they felt their office was organized and what problems they had in using it.

Individual differences

To illustrate some of the rich detail involved in how people organize their office work spaces, I briefly describe two of the cases I studied, including maps of their office organizations. The two cases were selected to represent two important kinds of office organization that stood out in this study: (1) Michael[1], a purchasing agent, had a "neat" organization that relied heavily on files and precisely organized piles, and (2) Kenneth, a research scientist, had a "messy" organization that included many very loosely characterized piles.

A rough analysis of all the interviews suggests that people with "messy" offices that include lots of "piles" have more problems finding information and remembering tasks to be done than people with "neat" offices.

[1] The names and initials of all the interviewees have been changed.

Major units of desk organization

As suggested above, one of the most salient features of the way people organize their offices is that some information is stored in "files" and some in "piles." *Files* are defined as elements of information that have explicit titles and are arranged in some logical order such as alphabetical or chronological. A folder in a file drawer is thus a prototypical example of a file. *Piles* are elements of information that have no explicit title and no logical arrangement (though they must have some physical arrangement). Piles of paper on a desktop are prototypical piles since they are not titled and, in general, are not systematically arranged in relation to each other.

Major units of desk organization

As suggested above, one of the most salient features of the way people organize their offices is that some information is stored in "files" and some in "piles." *Files* are defined as elements of information that have explicit titles and are arranged in some logical order such as alphabetical or chronological. A folder in a file drawer is thus a prototypical example of a file. *Piles* are elements of information that have no explicit title and no logical arrangement (though they must have some physical arrangement). Piles of paper on a desktop are prototypical piles since they are not titled and, in general, are not systematically arranged in relation to each other.

Functions of desk organization

It is no surprise that people organize their desks in part so that they can find things. But perhaps the most important insight from my study of desk organization was that, in addition to this *finding* function, an equally important function of most desks is *reminding*. Much of the information that is visible on top of the desks and tables in most offices is there to *remind* the user of the office to do something, not just to be available when the person looks for it.

A number of respondents made explicit remarks about the importance of reminding in determining what information was on top of their desks (e.g., "If I don't put it here where I can visually see it, I won't do it." (M.P., 10/27/81)). Also, a rough analysis of the interviews and photographs of the offices suggests that about 67% of the piles visible on tables, desks, and shelves were piles of things to do, presumably placed there to serve as reminders.

In addition to this evidence for the importance of reminding, a number of respondents commented on the difficulties of classifying information: "It's interesting how hard it is to characterize these things. . . It's almost like leaving them out means that I don't have to characterize them." (S.W., 10/12/81) "Well, see the hardest problem for me organizationally is deciding what the categories are and what category something is in" (K.H. 10/16/81)

Implications for designing computer-based information environments

Finding

Computer systems can help simplify both the mechanical and cognitive processes required for storing and retrieving information by: (1) allowing users to *defer classification* in the same way they do on desktops by storing the information in a physical location (e.g., in a "pile" on a simulated desktop) without having to explicitly title the information, (2) *automatically classifying* information as much as possible (e.g., based on access date, author, title, etc.), (3) allowing the same document to be easily put in *multiple categories*, and (4) allowing users to retrieve information based on more than one retrieval key at a time.

Reminding

Reminding is a subtle, but very important, aspect of desk organization. Failing to support this function can seriously impair the usefulness of electronic office systems. By explicitly trying to facilitate this function, computer-based systems may become even more useful. For example, if users classify the documents they want to be reminded of in terms of priority, then the system can use this priority to determine the *frequency* with which the documents are automatically displayed or their *size*, *location*, or *color* on the screen. The system can also automatically increase the priority of some items over time (e.g., important unanswered letters) while automatically decreasing the priority of others (e.g., unread journal articles).

Conclusions

One of the most salient features of the way people organize their offices is that some information is stored in "files" and some in "piles." Offices that include many "piles" appear to be "messy" and may also be less useful to their occupants. What, then, are the forces that lead to the creation of piles? There appear to be four:

(1) the mechanical difficulty of creating labelled file folders, binders, etc.--especially if multiple levels of classification are desired.

(2) the cognitive difficulty of creating appropriate categories and deciding how to classify information in a way that will be easily retrievable.

(3) the desire to be reminded of tasks to be done, and

(4) the desire to have frequently used information easily accessible.

Paradoxically, these forces can be self-defeating. As more and more information is stored in loosely characterized piles, it becomes harder and harder to find things, and items that were placed to be visible as reminders become covered up and no longer serve that purpose.

Electronic office systems can help with these problems by simplifying the mechanical processes of filing information, and by providing "intelligent" aids for categorizing and retrieving information and for reminding about things to be done. At the same time, these systems should continue to provide two of the functions of conventional desks: easy storage of loosely classified information and convenient use of visible reminders.

SESSION 4

PANEL: EVALUATION OF OFFICES

Chairperson: William Harless
Bank of America
Integrated Office Systems Division

The purpose of this panel is two-fold: 1) To clarify some of the social and behavioral issues in the evaluation of office information systems (OIS); and 2) to promote interaction between a few of the leading social and computer scientists in the field of OIS.

Inherent in the OIS setting is the direct association of non-technical people with sophisticated computer terminals. In the early stages this meant clerical staff using word processors. But currently OIS implementors are focusing on decision making processes of "knowledge workers" (including executives). By doing so these implementors find themselves face-to-face with the complex sociological, psychological, and technological problems of person-machine interaction. Issues abound - some subtle, some not so subtle, and social scientists are gathering in scores to get involved in the movement.

The second purpose of this panel is to promote interaction and, hopefully, a more complete integration between social scientist and computer scientist. This interaction will occur in light of a prevailing contemporary problem in the field - that is, that to date the issues and research results have tended to be platitudinous.

Perhaps the most common platitude is that the office is a socio-technical system. This perspective could guide research by defining a broader scope of investigation than either the social or technical scope would provide. This does not appear to be the case. One fundamental reason seems to be related to the type of research strategies so often used by the social scientists. There are numerous trappings and limitations of research which depends upon interviewing, unstructured observations and questionnaires. While such sources of data are necessary in the office setting, it must not be forgotten that they represent only a fragment of the information required to understand that setting. Methods must be developed and used which provide concrete measures of overt behavior.

Computer scientists on the other hand continue to abstract the human behavior in offices into deterministic formalisms. This approach is insufficiently subtle in that humans are obviously eliminated from the system. In fact, the very name of the conference, "office information systems" connotes that the people are not involved. At least the word "automation" has been set aside but, unfortunately, the concept most representative of the needs in offices, "augmentation", has not been widely accepted. It is revealing that we exist in such an embryonic stage of integration that we have not yet been able to adequately name ourselves. Nevertheless, if we view the office in its social context, including the society, the working life, individual morale and motivation, organizational and interpersonal relationships, we will have sufficient breadth of scope to permit valid evaluation of the office systems technology is transforming.

Clearly a partnership must form between social and computer scientists and intellectual integration must occur. We must be assertive about how a combination of social and technical evaluation methods would be implemented. Thus, the variables of both technology performance and human performance must be determined and

measured. Together we must identify the sources of spurious causality and control them. This implies the creative application of scientific method as much as possible in the complex office environment, including pretests, posttests, and even control groups. More importantly, it requires the evaluation of the total socio-technical office system as a unit with a sufficient number of users to normalize their differences. Most importantly, it requires taking the risk to grapple with the overwhelming complexity of the entire office system, and a commitment to technology that will likely extend over several years of refinement and redesign. The persons on this panel have evidenced such a commitment and their results have left the world of platitudes.

Extending the Power of Programming by Examples

Giuseppe Attardi and Maria Simi

Istituto di Scienze dell'Informazione
Corso Italia 40
I-56100 Pisa
Italy

Abstract

Programming by Examples is one of the methodologies that has been proposed for supporting the development of office applications by non computer specialists. Programs are built by performing direct manipulations on objects visually represented on a display, simulating the execution of the program on exemplary data items. We distinguish two major approaches: a declarative approach (SBA by DeJong and Zloof) and a procedural approach (Tinker by Lieberman and Hewitt). We examine the computational power and the expressive convenience of the two systems. The procedural approach achieves full generality by allowing the user to explicitly introduce iterative or recursive control structures. The SBA system has built in control mechanisms, which avoids explicit use of such constructs, but it is correspondingly less powerful. We present an algebraic model of SBA programs which allows us to determine the expressive power of the SBA language. We discuss a more general control mechanism then the one embedded in SBA, based on homomorphic transformations, which allows to compute all primitive recursive functions. Further flexibility is obtained by coupling the PBE methodology with the knowledge representation language Omega.

This work was done while the authors were visiting the Artificial Intelligence Laboratory of MIT. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505 and in part by the Office of Naval Research under Office of Naval Research contract N00014-75-C-0522. This research was supported in part through a grant from Olivetti. The second author has been supported in part by a fellowship of the Consiglio Nazionale delle Ricerche.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0052 $00.75

1. Introduction

Each office environment has peculiar information processing requirements and specific methods for managing its work. Moreover the procedures and the rules which are to be applied are in continuous evolution. Often the knowledge about the structure of the tasks and the procedures to be carried out is never fully documented and accumulates in the office worker minds as their experience grows. One conclusion from such observations is that the workers in the office itself should be provided with the ability and the tools for programming their work stations. Current programming techniques are not suitable for the office worker for several reasons. First of all extensive training is required because the users must:

- have precise knowledge of the syntax and semantics of a specific programming language;
- master the use of a number of complex tools: editors, compilers, debuggers.

Since the creation and the execution of programs happen in two separate stages, programming is like working blindfoldly: when writing a program one must foresee the situation when the program will be executing. When a mistake is made in the program, it is very difficult to relate the bug with the source code which caused it. The fact that large amounts of time are often spent in debugging the program is often a consequence of these conditions.

Finally, it is very hard to modify or extend an already written program.

It is undesirable that an office worker be compelled to communicate with the workstation in the machine oriented terms which are required by current computer languages. The worker must be able to communicate in the language in which he thinks and must be able to develop programs in as painless a fashion as possible.

The few systems which replicate some model of activity familiar to the users, have achieved remarkable success. An example is the *VisiCalc*™ system, which embeds a simple model of tabular bookkeeping derived from the business world.

In the following sections we review some of the systems that provide programming facilities by means of examples.

2. Programming in the System for Business Automation (SBA)

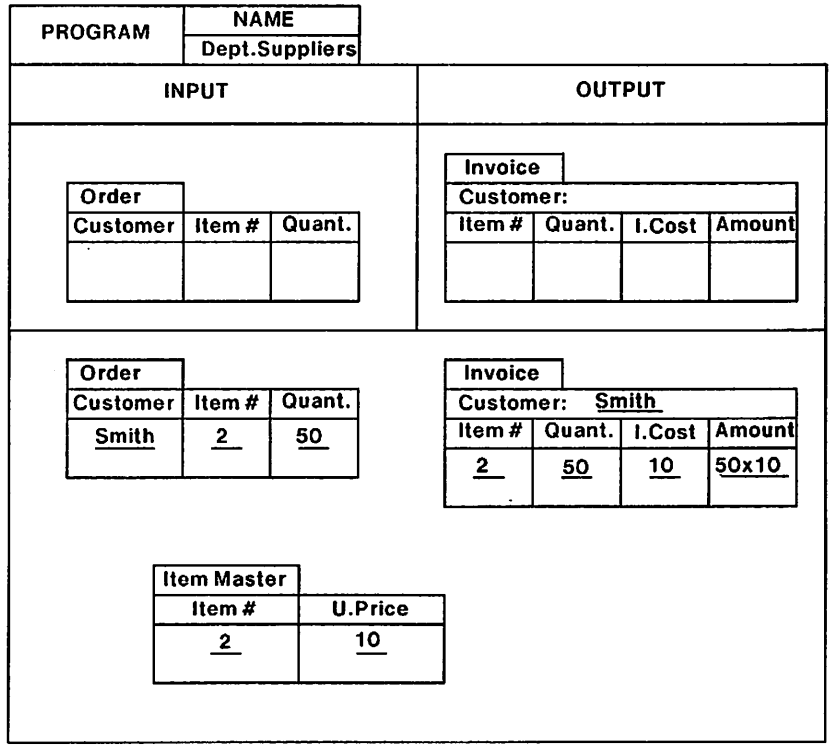
The programming language System for Business Automation [16] is a superset and natural extension of the Query-by-Example data-base management language [17]. It represents an attempt to combine

features from several domains, such as word processing, data base management, electronic mail, graphics and security within a single interface. Programming in SBA is performed by manipulating two-dimensional pictures of business objects.

A program in SBA expresses a relation between input data, specifically created new objects and actions to be performed on them.

Have the system to operate on concrete examples and to show the results of each step (retrieval or computation) as it is included in the program. The results of previous operations can be directly grabbed and used in successive steps. This eliminates altogether the need for "example elements", which become just internal names to the system for the results of certain operations. This is the approach of Tinker [12].

Figure 2-1: An SBA program



For instance in Figure 2-1, a program is presented for producing an invoice for each order. The item number from the order (let's say it is 2, an "example element" for what its real value is), is used to look up in the Item Master file, with a QBE query, for the price of such item (say 10). The amount field of the output invoice is the product of 10 and the quantity of the item ordered (which is, say, 50).

As a matter of terminology, we must remark that in the context of SBA and QBE the term "example" refers to an "example element", as opposed to a "constant element". An example element corresponds to what is elsewhere called a *variable*, which gets bound to some specific value during a pattern matching or unification process.

A potentially dangerous drawback of SBA is the fact that there is no guarantee that the values which will be bound to such example elements during program execution will be of the correct type as suggested by their names. For instance, in our example of figure 2-1, there is an example element 50 in the quantity field of the order form, which is multiplied with the price 10 in the amount field of the invoice form. There is no way to guarantee or check during program writing that the value of such quantity will actually be a number on which the multiply operation is meaningful.

There are two solution to this problem:

- The data base does not keep just raw data, but also semantics of such data, so that each data is described in terms of its type, each form has a description of the type of all its fields and of possible relation among these fields. This is the approach of the Omega description system for knowledge representation [3].

Both these approaches are included and integrated in the system proposed below.

2.1. The Power of SBA

The major limitation to the expressive power of the language of SBA is the absence of iteration constructs. A built in mechanism is provided that allows to operate on a batch of forms, operating on each one separately. Therefore programs can operate only on local data, available in the form itself. A few global operators are provided like SUM, AVG, COUNT.

In SBA it is possible to express conditions, but these are not used to alter the control flow, but rather to specify alternative results of the transformation under different conditions.

As we will see after we have sketched an algebraic semantics for the language, these constructs of SBA provide the power to express only a proper subclass of primitive recursive functions.

Of course, the limitations of SBA are also the reasons for its convenience and simplicity. Programs can be written without having to introduce explicit control information (iteration or recursion). Moreover there is no need to introduce program variables to store temporary results (accumulators, etc.).

Our goal in this paper is the design of a more powerful programming system which still retains most of these advantages.

3. Programming by Examples

At its simplest, programming by example is just recording a sequence of commands to a system, so that the sequence can be played back at a later time, to do the same or a similar task. The sequence forms a program. The user tells the system, "Remember what I am doing". The system executes the user's commands normally, but also remembers them.

The user has created a program by giving an example of what he wants the program to do. However, the statements in his program are the same as the normal system commands. The user does not have to learn some programming language with its attendant syntax and semantics; instead he can do his programming in the user interface of the system, which he already has to know anyway in order to operate the system.

In addition, he has written the program by performing, concretely, an example of what the program is supposed to do. Since he can easily verify whether he did his example correctly, he can be fairly confident his program will work. He does not have to understand his program abstractly.

Certain industrial robots, like Unimation Unimate , can be programmed by example by guiding them through the task they are to perform.

Another case of programming by example occurs in the Keyboard Macro facility of the display editor EMACS [14]. An editor macro, or a program to perform a series of editing manipulations, can be defined in the following way: the user tells EMACS to start recording his commands. Then he performs a sequence of ordinary EMACS commands. Each of these commands is executed normally, which means, since EMACS is a real-time editor, that the effect of each command is immediately visible on the display. A special command is used to terminate the sequence defining the macro. The sequence can then be invoked as a single command, which is executed on the text starting from the current position of the cursor.

Also in this case no significant extra knowledge is required other then the EMACS commands with which the user is already familiar.

Programs built in this way are useful to automate the execution of a sequence of operations which are repetitive and without variations. The limitation of these systems is immediately evident: straight line programs without parameters are not very interesting.

Before we get into the analysis of more sophisticated systems for programming by examples, we must mention a feature of the mail reader ZMAIL on the MIT Lisp Machine [15], which leads into the direction that we will pursue later in this paper. ZMAIL has a facility for filtering messages from a file of messages into a new file. A filter is built by combining with logical operators a number of primitive predicates (like seen, answered, replied, before, after), other filters or user supplied predicates. The user creates the filter by selecting from a menu the primitive predicates and the logical operators. ZMAIL in the meantime constructs a program corresponding to the given specifications. Once a filter is defined, an operator called Map can be invoked to apply the filter to each element of a file, producing a new file with just the messages which passed the filter.

3.1. Tinker

Tinker is a programming system for LISP which aims to integrate program testing with program design, by exploiting the programming by examples methodology. With Tinker the user *teaches* to the system how to carry out a task. Like when somebody trainee an apprentice, he shows the system what to do in a number of representative cases.

With Tinker, program construction is done in the following way: the user chooses a concrete typical example of data for the procedure; he then performs, step by step, the complete sequence of operations on the example data. Intermediate results are kept in a menu, called history, and can be manually selected with a pointing device to be used in successive steps of the computation. The system records the sequence of operations and constructs the program.

The program is incrementally extended for each new example supplied. As more examples are supplied, the system is able to notice if the sequence of operations for a new example does not conform to the program generated from recording previous examples. This is interpreted as the need for a conditional in the program and the user is asked to supply a predicate which will make possible to distinguish the new case from the previous ones.

This method of introducing conditionals leads the user to supplying one example for each branch in the final program.

Since programming with Tinker has the flavor of *concretely doing* a sequence of actions as opposite to *abstractly defining* a program, this methodology has been called *concrete programming* [12]. Operating on concrete data allows the user to see the effects of his program as he builds it; with this method the dichotomy between running and writing programs is partially dissolved.

Tinker provides the full power of traditional programming languages. It allows to define procedures with parameters. Conditionals and general recursion are available as control structures.

3.2. Halbert's System

Dan Halbert has built an interesting system for programming by examples which runs on a prototype of the Xerox Star professional office workstation. The Star workstation is part of the Star Information System, which consists of user workstations and one or more file servers, printers or communication gateways. The users and the severs processors are connected with an Ethernet local network. The Star workstation provides a number of utilities, for editing text and graphics in documents, print and file documents, send and receive documents as electronic mail, do record processing, deal with electronic forms, write application programs that process forms or documents.

An important characteristic of the Star is its uniform and simple graphical interface that allows most operations to be performed by selecting objects on the display and invoking commands from menus or from the keyboard. The screen in fact displays what is called a *desktop*, where small graphic icons are visible, representing *documents*, *folders*, *file cabinets*, *printers*, *in-boxes*, *out-boxes*, *record files* and so on.

Because of the richness of the operations that can be performed manually on such universe, the programming by example metaphore seems particularly appropriate in this context.

Halbert's system [9] allows simple command recording and also provides mechanisms for allowing the user to generalize programs and introduce iteration constructs. The system synthesizes a program in the programming language CUSP, which is provided in Star to write procedures specific to particular applications. CUSP can manipulate icons, text, table and form fields and other objects in Star. CUSP has an English-like syntax, as the following example, taken from Halbert's

thesis, shows:

```
Set Table.Net to Sales.Gross - Overhead;
ForEach Document D in Reports do
 Copy D to the Folder whose name is Audits;
end;
```

Halbert's system records steps being performed on real data, in the standard command interface of the Star. In order to distinguish between the constants and the parameters of the program, some commands are provided for telling the system when some data must be generalized into a parameter. Three kind of generalization are allowed:

same name
same place
ask for a similar object

The first means that the parameter will be designated by a name, which is taken as the same name of the example item. The second case is when the parameter is designated by its position on the screen. The third will allow the user to intervene during the execution of the program to select the appropriate argument.

The situation here illustrates one fundamental problem for programming by example systems, i.e. how the system can interpret the intentions of the user from the commands he issues. We will return to this question later, when we will advocate the integration of the methodology with more sophisticated descriptive capabilities.

In Halbert's system iteration can be introduced by invoking the command **For each do ...** while recording a program. The iteration is performed on a set of objects which is determined as a generalization from an example icon. In this case the iteration set is made up of all icons of the same type as the example icon, which are in the same container (file or folder). The body of the iteration is closed when the command **... end for each** is issued. The kind of generalization performed in this case by the system seems to have been chosen to reflect the structure of the iteration construct provided in CUSP.

Halbert's system still lacks the vital ability to introduce conditionally executed statements or expressions.

For systems like the Star, where the interaction is highly visual, programming by examples seems much more natural then the ordinary methods of program writing. Even with its expressive limitation, Halbert's system is claimed to be valuable in building interesting application programs.

4. Comparison Between the Previous Systems

While the styles of Tinker and Halbert's system are similar, these two systems differ more significantly from SBA. Tinker has a more procedural flavor, since with Tinker one performs a sequence of steps that will carry out the intended task. SBA is more declarative, since it allows to express a relation between the inputs and outputs of the program, i.e. how the result is built in term of the argument data. The order or the actual steps to compute that result are not specified and are left to the system, which is free to select the most appropriate or efficient way of computing it. And in fact SBA exploits this possibility by selecting an order which reduces the number of accesses to the data.

There are two other, perhaps minor, differences:

- SBA deals with collections of data, rather then single data items;
- examples are usually extracted from the data base, rather then supplied by the user.

Obviously SBA has major limitations: since it has no iteration nor recursion constructs, many algorithms cannot be expressed. On the other hand Tinker has complete generality since it allows full recursion and use of conditionals. Halbert's system is half way between the other two: it deals with real data as SBA, extracted from the Star *desktop* universe, it records and executes commands as in Tinker, but has only a limited form of iteration and no conditionals.

The question that we will address in the following sections is whether it is possible to extend the mechanisms of SBA in order to obtain a more powerful formalism while still retaining its simplicity.

5. Algebraic Foundations

We present some terminology and results that will be used in describing the semantics of SBA and studying its expressive power. The results of this section will also be the basis for the design of the more powerful system proposed in the paper. The idea of programming by homomorphism and most of the results on which it relies where discussed first by Burstall and Landin in [7]. Properties of homomorphism and their use in program improving transformations are investigated in [2].

5.1. Algebras

A many-sorted algebra consists of an indexed family of sets (called the carriers) and an indexed family of operations defined on Cartesian products of those sets. The index set for carriers is called the set of sorts, and might be, for example {Integer, Boolean, List}. An algebra A of this kind would have three sets, $A_{Integer}$, $A_{Boolean}$, A_{List}, together with some operations such as CONS : $A_{Integer} \times A_{List} \rightarrow A_{List}$, and NIL : $\rightarrow A_{List}$. This last example shows that we can consider the constant NIL as a zero argument operator which returns the empty list.

We will discuss the case of one-sorted algebras, since the generalization to the many-sorted case can be done along the lines of [6].

By a set of *operators symbols with arity* we mean a set S, whose elements are called operator symbols, together with a function $S \rightarrow N$ which tells for each operator symbol the number of arguments that the corresponding operator needs. By a *S-algebra* $\langle A, \Phi_S \rangle$ we mean a set A, called the *carrier* of the algebra, and an operator set Φ_S, which is a family of operators $\{\varphi_s \mid s \in S, \varphi : A^{arity(s)} \rightarrow A\}$.

By the *closure* of a set X with respect to Φ we mean the least set A which contains X, and such that for each φ of arity n, if $a_1, \ldots, a_n \in A$, $\varphi(a_1, \ldots, a_n) \in A$. We will denote such a closure by *Closure* (X, Φ). X is called a *generator* set for A. The closure of an algebra corresponds to the notion of *initial algebra* in [8].

5.2. Homomorphisms

Given two S-algebras $\langle A, \Phi_S \rangle$ and $\langle B, \Phi'_S \rangle$, an *homomorphism* is a function $h : A \rightarrow B$ such that for each s in S of arity n:

$$h(\varphi_s(a_1, \ldots, a_n)) = \varphi'_s(h(a_1), \ldots, h(a_n)))$$

So a homomorphism is a mapping which preserves the operations. We review some of the fundamental properties of homomorphism that will turn out useful to provide convenient way of defining them.

> Lemma 1: (*Unique extension*) If A = *Closure* (X, Φ), B = *Closure* (Y, Φ) and f is a function $f: X \rightarrow B$, there exist at most one homomorphism $h: A \rightarrow B$ such that for all $x \in X$, $h(x) = f(x)$.

This property provides a concise way of specifying a homomorphism between two algebras A and B. The homomorphism can be defined by

giving:

- a transformation f: X → B from a generator set X for A into B.

- a mapping F: Φ → Φ' which associates to each operator on A an operator on B with the same arity.

The defined homomorphism will be the unique extension of f which preserves the operations as mapped by F. Such a definition requires a justification that such a homomorphism indeed exists.

The following result provides a sufficient condition.

> Lemma 2: If each element of an algebra A can be generated in a unique way from elements of a generator X for A, then for any function f: X → B there exists a homomorphism that extends f.

Such condition will be fulfilled in the sequel because the operators of our algebras will be data type constructors, where different type constructors yeld different objects.

Another relevant property of homomorphisms is about their composition:

> Lemma 3: If h : A → B and g : B → C are homomorphisms, then so is their composition h g : A → C.

This result is significant because it suggests a way to optimize programs written as homomorphisms, so that such programs will not be penalized in term of execution efficiency. In fact the two nested recursions of a composition of homomorphism can be transformed by a simple manipulation into a single recursion.

5.3. Operatorswith Identity

Suppose that a binary operator φ of an algebra A has a right identity e, such that for all elements a of A: φ(a, e) = a.

A homomorphism h which maps φ into φ' must map e into the corresponding identity e' of φ'. In fact h(a) = h(φ(a, e)) = φ'(h(a), h(e)).

Such property of identity are very useful when defining homomorphisms, since there is no need to specify how the identities of binary operators have to be mapped.

For instance, suppose we want to define a homomorphism length which counts the number of letters in a word. The algebra of strings is defined by the closure of the set of characters A, under the binary operator concat, with identity the empty word λ. The range of length is the algebra of integers, with the sum operator and identity 0. To specify the homomorphism, one needs only to say that characters are mapped into 1, and that + will correspond to concat. Therefore

$$h(abc) = h(concat(a, concat(b, concat(c, \lambda)))) = +(h(a), +(h(b), +(h(c), h(\lambda)))) = +(1, +(1, +(1, 0))) = 3$$

Because of the convenience of identities, we will use a convention that will allow to manifacture one for many kinds of operators. Suppose we have a binary operator f: A x B → B. We will extend f to a function f* with an arbitrary number of arguments as follows:

$$f^*(x_1, \ldots, x_n) = f(x_1, f^*(x_2, \ldots, x_n))$$
$$f^*(x) = x$$

Since by the first equation f*(x) = f(x, f*()), it turns out that f*() will have to be a right identity.

We will apply this device for extending the binary constructor for lists (CONS) into the n-ary constructor (CONS* or also LIST), and similarly the constructor for adding an element to a set into the constructor of an n-element set. So for instance:

$$LIST(a, b, c) = CONS(a, CONS(b, CONS(c, LIST()))) = CONS(a, CONS(b, c))$$

A peculiar consequence of this choice will be the fact that there will be no distinction between an element and a list with just that element. But this will make no noticeable difference if we restrict ourselves only to functions defined as homomorphisms. In fact, suppose h is such a function:

$$h(CONS(a, CONS^*())) = CONS'(h(a), CONS'^*()) = h(a)$$

then h gives the same result on the element a then on the list with one element a.

A semigroup is an algebra with an associative binary operator which has an identity (both left and right).

5.4. Primitive Recursion via Homomorphism

Any primitive recursive function can be programmed in terms of homomorphisms. We show just the simple case of unary functions from natural numbers into (functions over) natural numbers.

The algebra of natural numbers is defined as *Closure* ({0}, {Succ}), where Succ is the successor function.

A primitive recursive function defined as:

$$f(0) = \lambda y \,.\, g(y)$$
$$f(Succ(n)) = \lambda y \,.\, h(f(n)(y))$$

is a homomorphism where:

0	is mapped into	$\lambda y \,.\, g(y)$
Succ	is mapped into	$\lambda z \,.\, \lambda y \,.\, h(z(y))$

6. Algebraic Semantics for SBA

We can provide a model of SBA programs in terms of the algebraic concepts introduced above.

Consider a SBA program, which operates on a batch of forms. The SBA programs specifies a transformation between each of these forms and the corresponding output form.

The program can be modeled as operating on a set S of forms, applying a transformation T to each of them and producing the set V of transformed forms. This operation is an instance of a homomorphism between two algebras of sets. The algebra of sets is actually a semigroup with the set of all forms F as a carrier, a binary associative operator ∪ and a constant {} which is the identity for ∪. Since the empty set has to be mapped into itself, the only thing that needs to be defined is the mapping between the individual elements of the set.

The transformation T is any one that ca be expressed in terms of data base queries, primitive or defined functions and conditionals. The operator ∪ is however mapped by default into the operator ∪ itself. This accounts for one fundamental limitation of the principal control structure of SBA. Since homomorphism provides the power for expressing all primitive recursive function, SBA restricted form of homomorphism is limited to a proper subset of primitive recursive functions.

Among the operation that cannot be performed with this construct are functions that compute the sum or the average of quantities in the input form. For some of these, SBA provides explicit primitives like SUM, AVG, COUNT. As we will show later, all of these could be easily programmed, if the general form of homomorphism would be allowed.

SBA provides a few more constructs that mitigate this limitation. Assignment, sequencing and goto constructs are in fact provided, to allow a more traditional style of programming.

7. Raising the Level of the Conversation

The metaphore applied in programming by example systems like Halbert's is to allow the user to perform any command or operation provided by the command interface on real specific objects. This raises a problem for the system on how to interpret these operation in the general setting of the program being built. Suppose the user moved the cursor two lines up in the text of a document: did he really mean that, or rather he wanted to go to the first line of the previous paragraph, or to go to line 37, which happened to be two lines above the current one? Obviously no system can always be able to guess the intentions of the user when he issues such commands. We suggest that the only way out from this situation is to base the conversation between the user and the system on more semantic terms. This means to identify objects in term of their attributes, properties or relationships with other objects. This will require the use of a descriptive language to interact with the system when building programs more sophisticated then the one used in normal interaction.

A second suggestion is to use particular programming constructs, where certain objects assume specific roles. Selecting such an object would be interpreted as specifying an object that fullfills such role. For instance, in defining a function with one parameter, the user would supply an example object as an argument for the function on which to show what it should do. Within the scope of the definition, selecting that object will mean selecting the argument to the function.

These two criteria have been used in the design of the system that we present in the next section.

8. Programming by Homomorphisms and Descriptions

We present the ideas for a system for programming by examples, which extends SBA in two directions, by introducing a full general homomorphism construct and by employing a knowledge base of descriptions as its domain of operation. For these reasons we call the system PHD (Programming by Homomorphisms and Descriptions).

In this section we give a first example of programming by homomorphisms. In addition we will present a convenient interface for program definition that we hope will substantiate the claim that programming based on this idea can be quite simple.

Like in Tinker, programming will be done interactively, executing operations on concrete examples. A suitable pointing device should be available for object selections. Operations are performed on graphical objects, similar to the forms of SBA, consisting of a name and a number of attribute fields.

8.1. The Omega Description System

PHD uses an underlying description system, called Omega [10, 3, 4], for describing the objects being manipulated and their relations with other objects.

The basic notion in Omega is the notion of description, for example:

(*an* Item (*with* unit.price $2) (*with* supplier Adams))

describes all the items with unit price $2 which are supplied by Adams.

Descriptions can be combined by means of description operators and also nested within other descriptions, and they can contain variables, like =p in the following example:

```
(an Item (with in-stock (< =p))
         (with reorder =p)
         (with supplier
           (a Supplier (with location (Boston or Cambridge)))))
```

Graphical objects on the screen are just external representations of Omega descriptions.

The primitive statement in Omega, called *predication*, is obtained by relating two descriptions with the transitive relation *is*. An example of predication is the following:

(*an* Item (*with* item# 20)) *is* (*a* Hardware-item)

meaning that each item with item# 20 is an item of type hardware. Intuitively *is* corresponds to the inclusion relation between sets.

A data base can be represented in Omega as a set of assertions of the kind:

**2F-12 *is* (*a* DiskCartridge (*with* manufacturer Memorex)
(*with* in-stock 15))**

**GS-05 *is* (*a* DiskCartridge (*with* manufacturer 3M)
(*with* in-stock 3))**

**DF-33 *is* (*a* FloppyDisk (*with* manufacturer Maxell)
(*with* in-stock 50))**

(*a* DiskCartridge) *is* (*a* MagMedia)

(*a* FloppyDisk) *is* (*a* MagMedia)

With Omega one can build a network of descriptions, on which semantic information retrieval can be performed on the basis of properties of objects.

8.2. An Example of Homomorphism

As a first simple example we show the definition of a function to perform the following task:

Compute the total amount due from a collection of invoices, summing up the amounts from the invoices which have not been paid.

We present the development of the program by means of a sequence of snapshots each one showing the result of one or more commands.

In the first step (fig. 8-1) the user issues a command to define a function with name Total-due with no arguments. A skeleton for the function appears on the screen with a field devoted to the result.

The next three steps consist in the retrieval from the data base of all invoices which have not been paid. The second step (fig. 8-2) is the creation of a new blank invoice form. The user prepares a description of the items he wishes to retrieve from the data base by means of the new form. He does so by filling in the field paid with the constant NO. In fig. 8-3, after the invoice has been selected, a query command is issued. The selected invoice is the argument to the query. The result of the query is a list of invoices, displayed in a succinct form.

Note how we avoid with the initial query the conditional testing if an invoice has been paid.

The important step is the calculation of the total. According to our methodology this amounts to defining the mapping which characterizes the homomorphism. The user will have to specify the mapping for the elements of the list of invoices and specify how the results have to be combined, i.e. which operator will be used in correspondence to the operator list. In the following step step (fig. 8-4) the command map over for the definition of a homomorphism is issued and the selected item (the list resulting from the query) is the argument to the map over command. In reply to this command the system adds a column to the form in which the list is presented, creating space for the mapping to be specified and shows in full the first element of the list of invoices, on which the user can immediately operate.

Figure 8-1: Step 1: Define function (*a* Total-due).

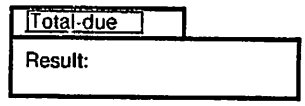


Figure 8-2: Step 2: Create (*an* Invoice), Fill in "NO".

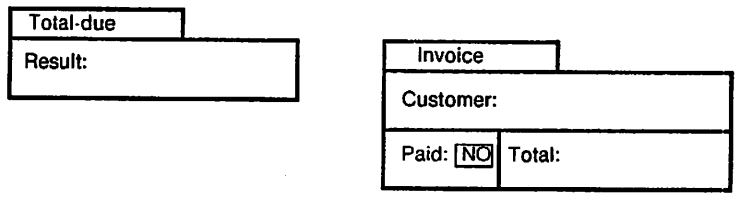


Figure 8-3: Step 3: Select invoice, query.

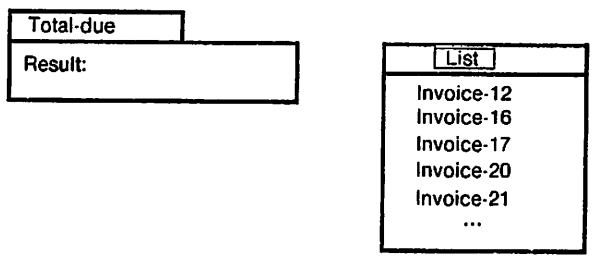


Figure 8-4: Step 4: Map over, Select total.

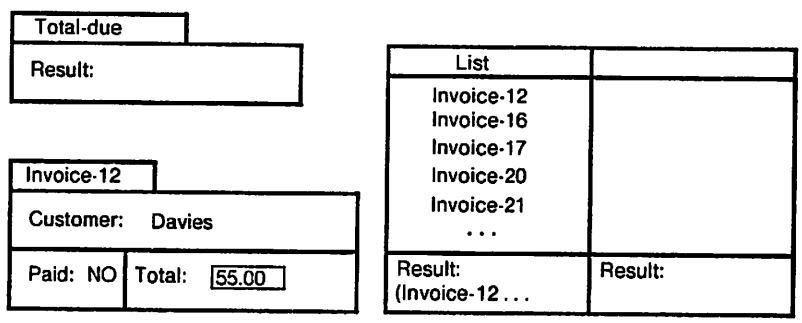


Figure 8-5: Step 5: Return, Proceed.

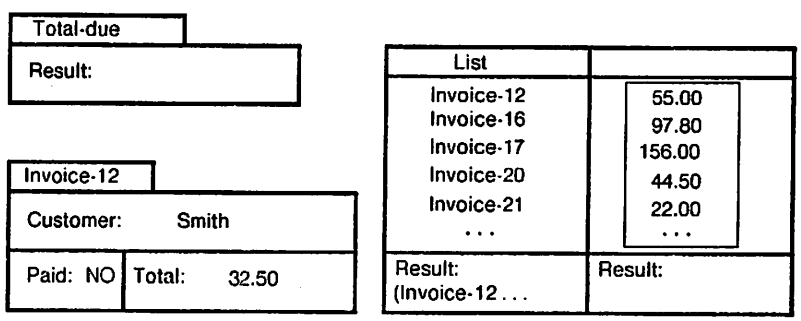


Figure 8-6: Step 6: Fill in "+".

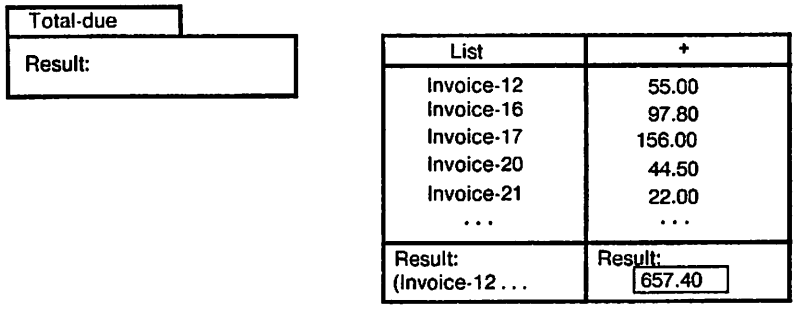
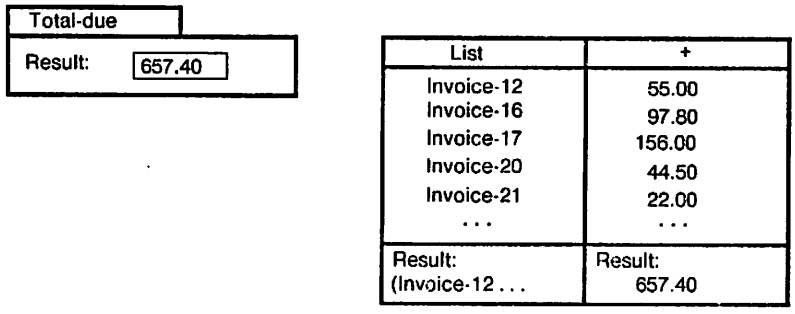


Figure 8-7: Step 7: Copy result, Return.



The user will have to specify the desired mapping just once on this concrete example of invoice, namely invoice-12. In this case all he does is to select the total field and issue the command return (fig. 8-5) meaning that the contents of that field is the result of the operation on the invoice. The result is placed in the column of results in correspondence to invoice-12. The system computes the same function on all the other elements of the list creating a list of numbers.

In fig. 8-6 the user specifies how to combine this list of results: to compute the total due the correct operation is to add all the element of the list, so the symbol "+" is filled in the operation field. The systems computes the result and shows it in the result field.

The last step (fig. 8-7) is simply to copy the result from the homomorphism table to the result field of the Total-due function.

In the example we can see the convenience of having used the convention established in the section on Operators with Identity, for dealing with operators with an arbitraty number of arguments. This gives the freedom of handling tables for list with no elements, where the result of the function will be taken to be the identity, lists with a single element, where the result will be the element itself.

Notice that knowledge of identities of operators is necessary even in systems like *VisiCalc*, to determine the result of operations which involve fields in a form which have not been filled. It is appropriate in fact to assume that they are filled with the identity of the operator.

8.3. Programming with Filters and Homomorphisms

Homomorphisms are a suitable control mechanism for building programs with loops. As we saw in the examples, a homomorphism embeds both recursion and conditionals. While in many cases the conditionals provided inside the homomorphism are sufficient, the more general mechanism of *filters* proves to be very useful.

A filter is just defined by a set of Omega descriptions, possibly including some common variables. Given a filter $F = \{\delta_1[x, y], \delta_2[y, z], \delta_3[z]\}$ the operation Query F returns the list of sets $\{e_1, e_2, e_3\}$, consisting of descriptions of individuals e_1, e_2, e_3 such that, for some individuals a, b, c:

e_1 *is* $\delta_1[a, b]$
e_2 *is* $\delta_2[b, c]$
e_3 *is* $\delta_3[c]$

The following example illustrates the use of filters and a general programming methodology. The methodology is based on building programs out of two parts: a filtering operation and a homomorphism. The program has to accomplish the following task:

Compare requisition forms with receiving report forms: for those requisitions which have been fulfilled create a voucher form with the total amount due

As in the previous example, the user starts with the command "define a function"; the function name is Voucher-Preparing.

Step 2 and 3 are the preparation of a query to the data base. The items that have to be retrieved are pairs of forms, the first a requisition and the second a receiving report, constrained as specified. More precisely, by filling =R in the Req# fields of both the requisition and the receiving report forms, the user expresses the requirement that those fields have the same contents. Names prefixed by "=" are variables in the Omega notation.

Before the query command is issued both the requisition and the receiving report are selected meaning that the query is on both forms. The descriptions underlying the two forms act as a filter on the data base, in the sense explained before.

Figure 8-8: Step 1: Define function (*a* Voucher-Preparing)

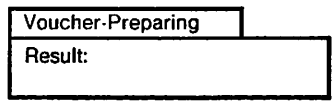


Figure 8-9: Step 2: Create (*a* Requisition), create (*a* Receiving-report)

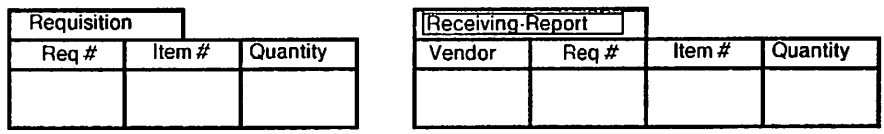


Figure 8-10: Step 3: Fill in and select requisition and receiving report

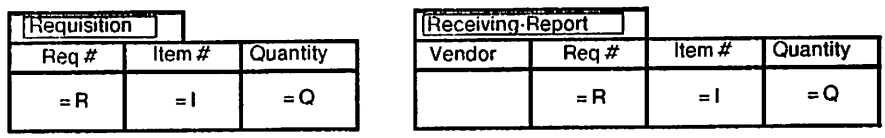


Figure 8-11: Step 4: Query, Map over

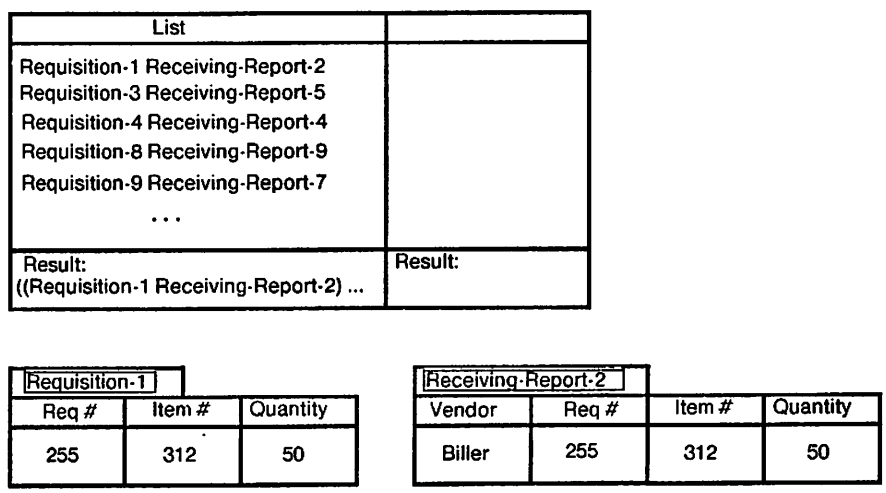


Figure 8-12: Step 5: Copy item # from requisition-1, select item-master

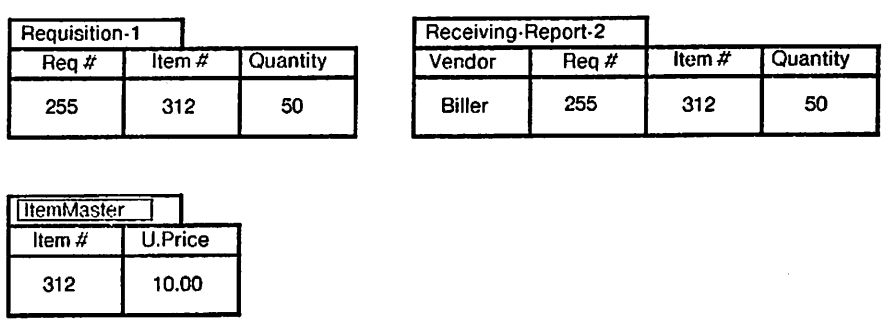


Figure 8-13: Step 6: Compute U.Price x Quantity

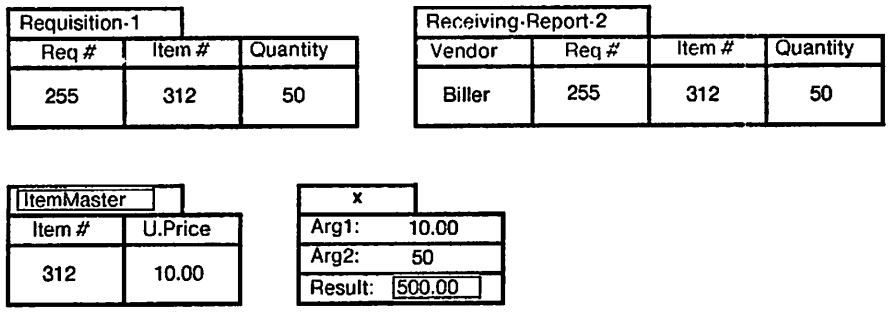


Figure 8-14: Step 7: Create voucher, copy vendor, copy result

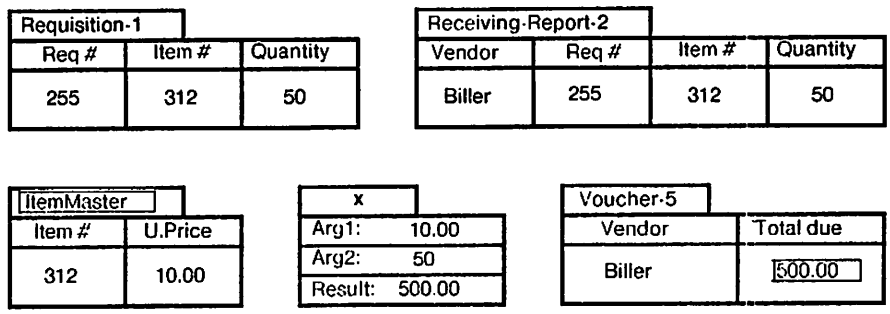
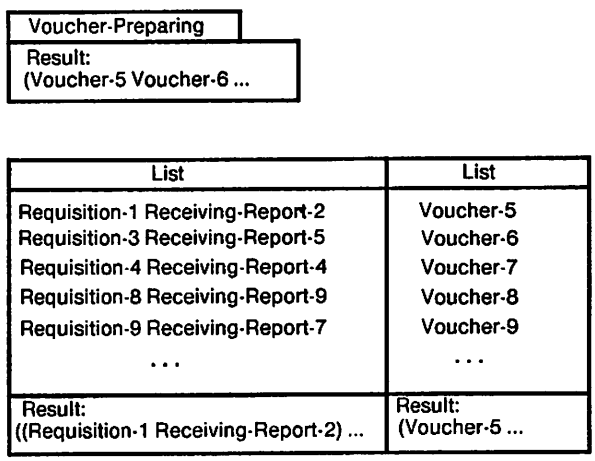


Figure 8-15: Step 8: Select voucher-5, Return, Fill in "List", copy result



The result of the query is a list of pairs of forms with the desired constraint. The map over command is issued on this list (fig. 8-11) and as a result requisition-1 and receiving-report-2, the first pair of forms in the list, are shown for the user to operate on.

At this point the user performs another query to the data base for retrieving the unit price of the item whose item# is specified in both forms (fig. 8-12). From the result of the query he can extract the unit price of the item number 312, and compute the amount due by multiplying it with the contents of the field quantity (fig. 8-13).

The result of this operation, is copied into a newly created voucher form. The name of the vendor is also copied into this form from the vendor field of receiving-report-2 (fig. 8-14). This voucher is returned in correspondence to the pair <requisition-1 receiving-report-2> in the homomorphism. The system will then complete the mapping for every other element in the list. The user will specify to take the list of these vouchers and return as the final result of the function (fig. 8-15).

Note how with the mechanisms of filters and homomorphisms nested loops can be avoided.

8.4. Scheduling a Meeting

We will consider now the problem of arranging for a meeting between a group of people in an office. We will write a program to find the hours in which all the people in a certain group are available for partecipating in the meeting. In this example we will see how conditionals can be introduced in a program.

We assume that all the members of the group maintain their agendas and the agendas are publicly accessible.

So we start defining a function, AvailableMeetingHour, with two arguments (Figure 8-16). The first argument will be the description of the group, and we supply as an example the group of managers in the sales department. The second argument is the period within which the meeting is expected to take place, and we provide as an example a two week interval starting on October 8th. So we create the description

(*an* AvailableMeetingHour
(*of* group (*a* Manager (*with* dept (*a* SalesDept))))
(*of* period (*a* TimeInterval
(*with* start 10/5/@9:00AM)
(*with* end 10/16@5:00PM))))

We then form the description of a business hour within the limits of the given period (Figure 8-17).

Now querying on that description, we obtain the list of all such hours (Figure 8-18).

For each of these hours (Figure 8-19), we want to look into the agendas of all the managers to see if they all have no engagements. We take the first hour (all the other will be dealt similarly by the homomorphism) and we create a description of the agenda of a manager referring to that hour (Figure 8-20).

By querying on this description, we get a list of the agendas at 9AM on October 5th (Figure 8-21). We start a nested homomorphism on this list. We pick up the value in the available field of one agenda and return that as a result. We let the system proceed similarly through all the other agendas, and then we combine all the results with the logical "and" operation Alltrue (Figure 8-22). Since in this case not all the people are available, we want to discard the hour we are examining. Therefore we return *Nothing*. The nested loop is concluded, so we let the system step through the second and third hour in the list. At this point we have a case in which all managers are available. Therefore this time we want to return the hour. The system now recognizes a situation differing from the previous ones, so it requires the user to fill a condition to distinguish the two cases. Here we can just indicate the result of the Alltrue operation as the value of the test (Figure 8-23).

Figure 8-16: Step 1: Define Function **(*an* AvailableMeetingHour**
(*of* group (*a* Manager (*with* dept (*a* SalesDept))))
(*of* period (*a* TimeInterval (*with* start 10/5/@9:00AM) (*with* end 10/16@5:00PM)))).

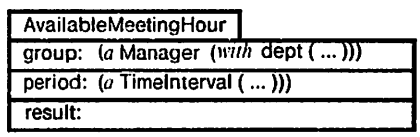
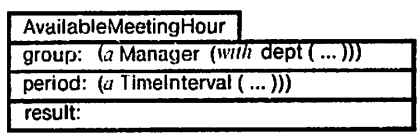


Figure 8-17: Step 2: Describe possible meeting hours.



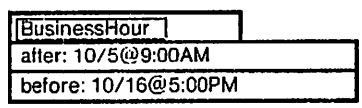
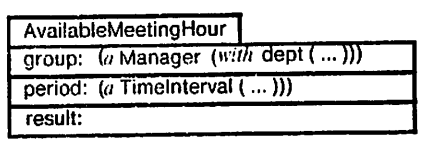


Figure 8-18: Step 3: Query.



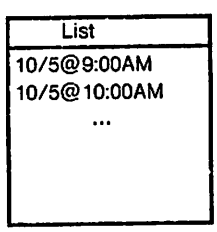


Figure 8-19: Step 4: Map over.

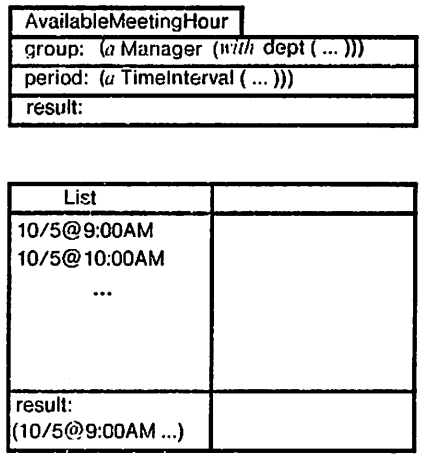


Figure 8-20: Step 5: Create description of Agenda, fill person, hour.

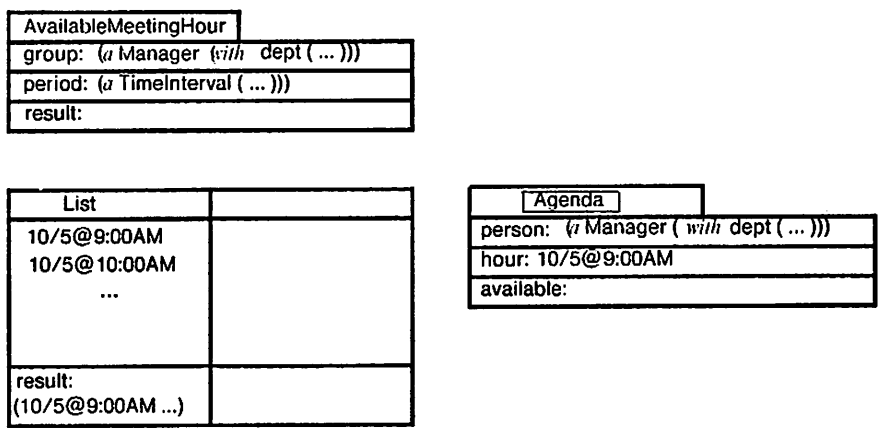


Figure 8-21: Step 6: Query the agendas.

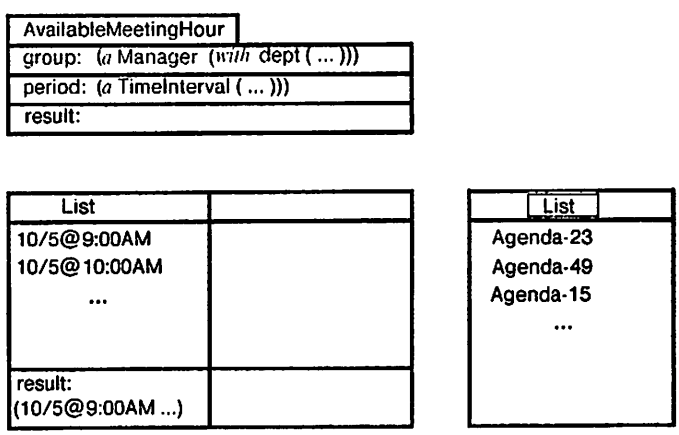


Figure 8-22: Step 7: Map over list of agendas.

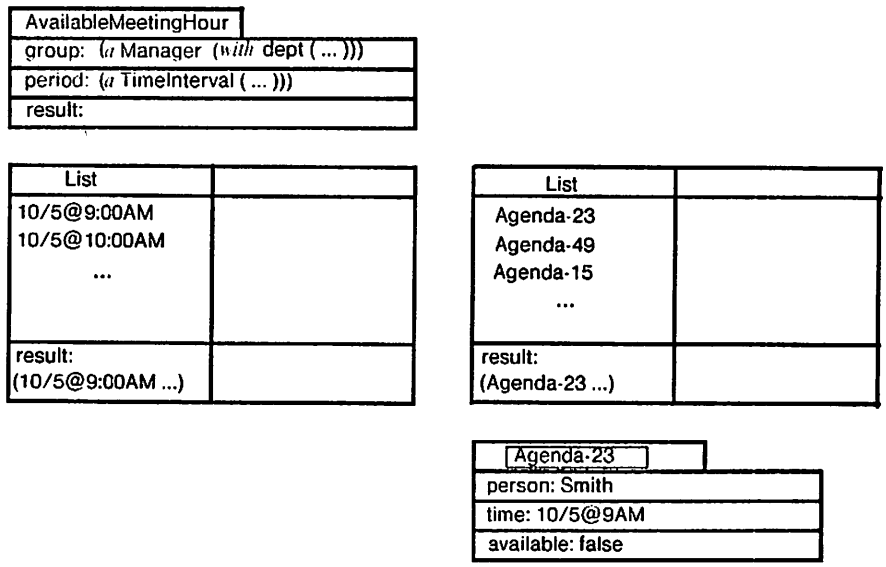


Figure 8-23: Step 8: Select false, Return, Proceed, Fill in AllTrue.

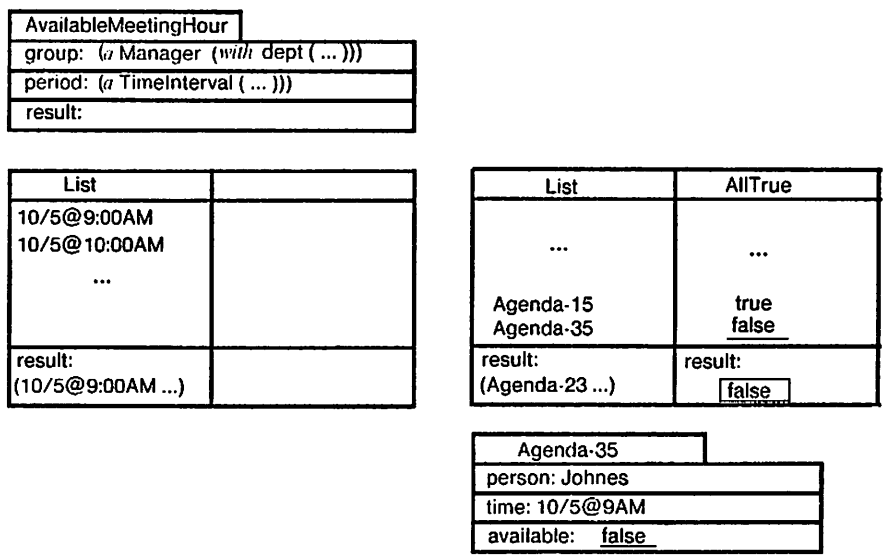


Figure 8-24: Step 9: Return *Nothing*, Step through other hours.

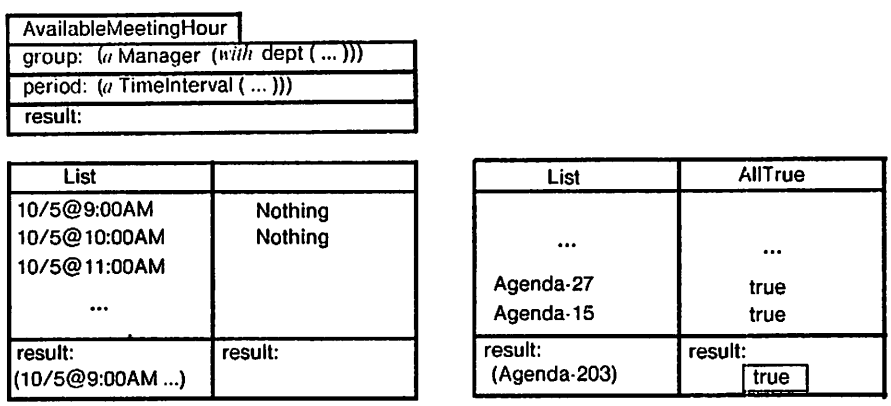


Figure 8-25: Step 10: Return hour from List, Fill in condition.

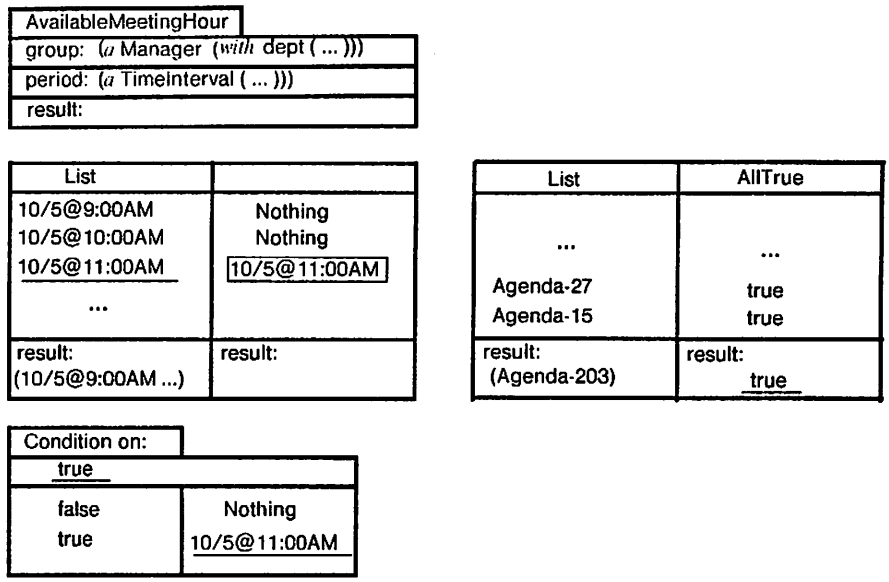
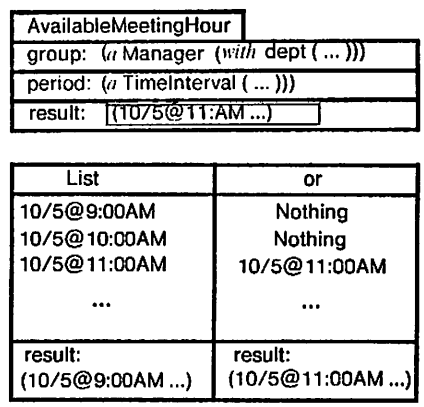


Figure 8-26: Step 11: Proceed, Fill in *or*, Copy result, Return.



From now on we can just let the system proceed, and when the process is completed for all the hours in the list, we indicate that we want to form the collection of the results by means of the description meet operator *or* (Figure 8-24).

In solving this problem we relied on a certain amount of knowledge that the system was supposed to have about times, dates, intervals and temporal relations. This is a domain where a flexible knowledge representation language is in order because of the intricate ways in which these concepts are connected.

8.5. Conditionals

In a function defined by homomorphism, there are two places where a conditional expression might have to be included: within the transformation mapping f on the generators X, or within one of the operators φ' in the target algebra B.

We have seen in the previous example how the first case can be dealt with.

The second case occurs when an operator which recombines the partial results of the homomorphism depends on the values of these results. A simple example is a function which inserts an integer i in the appropriate position into a list of integers in increasing order. Here the homomorphism transforms each generator of the algebra (an integer) into itself. The operator which recombines these elements is CONS, as long as i is smaller the current element. Otherwise it is λ x y . CONS (x, CONS (i, y)). When this situation occurs, the user will specify this operator by performing the two CONS operations involved. As in the previous case, the system will request a predicate to distinguish between the two alternatives. Such predicate will be embedded in the code for the operator.

9. Conclusions

Programming by homomorphism has already proven to be a valuable and practical programming methodology. This is testified by the Jackson methodology [11], which has been profitably applied in business data processing. The Jackson methodology is based on building programs whose control structure is patterned after the data structures on which they operate. This results in the construction of programs which are homomorphic transformations.

A significant advantage of the methodology is that well structured programs are created. These programs only access local information and are therefore easy to modify: extensions or minor modifications only require local changes which do not affect other parts of the program.

For the same reasons of locality of reference to data, homomorphic programs provide potential for lots of parallelism. For instance, when the program operates on a list or on a set, the transformation can be applied to all the elements of such structures in parallel. The properties of the construct guarantee that there will be no interference among those operations.

The programming language Awk [1], is based on a model which is similar to our proposal of using homomorphic transformations. A stream of input is processed by applying transformations throughout its elements. The elements in the file to which to apply the transformation are selected by pattern matching. An Awk program consists of pairs *pattern {action}*. For each line in the input stream which matches the pattern, the corresponding action is performed. Awk has proven a very succesful and useful tool in the UNIX users community. It has been used in a broad variety of tasks, ranging from budget preparation to telephone network planning. . Awk provides a convenient way of performing those many tasks which require minor transformations on already existing information, like selecting, rearranging and combining data. Awk is still a conventional programming language, in the sense that programs have to be hand coded and creating and testing a program are performed in two separate stages.

The examples presented in this paper only sketch the general idea for a system of programming by homomorphism. A practical system will include many more features, like options for stepping forward and backward through an execution; stopping to modify a program where appropriate; running a homomorphism either bottom-up or top-down.

There are also a few theoretical aspects that deserve further investigation. For instance, we conjecture that all recursive functions can be defined by homomorphisms if infinite data structures are allowed.

10. Related Research

The seminal work in programming by example was done by Dave Smith of Xerox in his Stanford Ph.D. Thesis [13]. Gael Curry did similar work in which the examples given by the user could contain abstract as well as concrete data values.

Bierman [5] developed a system, called *autoprogrammer* who synthesizes programs from traces of their execution on examples and is able to introduce loop expressions. A problem with this approach is that the user does not provide to system any suggestion about the structure of the program or the semantics of the problem. Therefore any new example supplied to the autoprogrammer may lead the system to create a completely different program from the one it had built so far. There is no way for the user to predict when he has supplied enough examples so that the correct program is synthesized.

11. Acknowledgments

The ideas presented have evolved from those discussed during the preliminar design of Tinker, in conversations with Carl Hewitt, Henry Lieberman and Ken Kahn. Dan Halbert provided insights into the relations between SBA and other approaches to programming by example. Massimo Massironi provided comments on an earlier draft. James Bair suggested the meeting scheduling problem as a test case for our methodology.

References

1. Aho, A. V., Kernighan B. W. and P. J. Weinberger. Awk - A Pattern Scanning and Processing Language. *Software - Practice and Experience* 9 (1979), 267-279.

2. Aiello, L., Attardi, G. and Prini, G. Toward a More Declarative Programming Style. In E.J. Neuhold, Ed., *Formal Description of Programming Concepts*, North Holland, 1978, pp. 121-137.

3. Attardi G. e M. Simi. Consistency and Completeness of OMEGA, a Logic for Knowledge Representation. Seventh International Joint Conference on Artificial Inteligence, Vancouver, Ago., 1981.

4. Attardi G. and Simi M. Semantics of Inheritance and Attributions in the Description System OMEGA. AI Memo 642, MIT, 1981.

5. Biermann, Alan W., and R. Krishnaswamy. Constructing Programs from Example Computations. *IEEE Transactions on Software Engineering SE-2*, 3 (September 1976), 141-153.

6. Birkhoff, G. and Lipson, J.D. Heterogeneous Algebras. *Journal of Combinatorial Theory*, 8 (1970), 115-133.

7. Burstall, R.M. and Landin, P.J. Programs and their Proof: an Algebraic Approach. In *Machine Intelligence 4*, American Elsevier, 1969, pp. 17-44.

8. Goguen, J.A., Thatcher, J.W. and Wagner, E. G. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. Research Report RC 6487, IBM T.J. Watson Research Center, 1976.

9. Halbert D. C. An Example of Programming by Example. University of California, Berkeley, 1981.

10. Hewitt C., Attardi G. and Simi M. Knowledge Embedding with the Description System OMEGA. Proc. of First National Annual Conference on Artificial Intelligence, Stanford University, August, 1980, pp. 157-163.

11. Jackson, R. *Principles of Program Design.* Prentice-Hall, 1970.

12. Lieberman H. and Hewitt C. A Session with Tinker: Interleaving Program Testing with Program Writing. Conference Record of the 1980 LISP Conference, Stanford University, August, 1980.

13. Smith D. C. Pygmalion: A Computer Program to Model and Stimulate Creative Thought. Stanford University, 1975.

14. Stallman, R.M. EMACS Manual for ITS Users. A.I. Memo 554, MIT, April, 1981.

15. Weinreb, D. and Moon D. LISP Machine Manual. MIT, AI Laboratory, July, 1981.

16. Zloof, M.M. and De Jong, S.P. The System for Business Automation (SBA): Programming Language. *CACM 20*, 6 (1977), 385-396.

17. Zloof, M.M. Query-by-Example: A Data Base Language. *IBM Systems Journal 16*, 4 (1977), 324-343.

An Actor-Based Programming System

Roy J. Byrd
Stephen E. Smith
S. Peter de Jong*

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT: A programming system is described with which applications are built by defining collections of communicating objects, called actors. The actor programming system provides a uniform environment in which distributed applications can be automated in a highly modular and efficient manner. The system's design is based on the formal theory of actors, with certain modifications made for the sake of efficiency. We describe our view of the actor system, and an implementation of that view. We also discuss applications built on, and contemplated for, the actor system.

1. Introduction.

This paper describes a programming system developed within the context of the System for Business Automation (SBA) project (de Jong (1980), Zloof and de Jong (1977), de Jong and Byrd (1980)). In SBA, "boxes" represent business objects like documents, files and memos. Initially humans operate on the objects. Gradually the business processes are automated by imparting intelligence to the objects through the 2-dimensional SBA programming language. Eventually one obtains an automated business system in which intelligent business objects communicate among themselves without human intervention. These systems are naturally distributed, and we envision a network of processing nodes representing the functional components of a business organization, with objects such as memos and invoices flowing between these nodes.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0067 $00.75

Instead of directly implementing SBA in PL/I, we decided to first develop a programming system which provides the primitives needed by SBA. Such a programming system should support intelligent objects which communicate via messages, the ability to create instances of objects which have common structure and behavior, and the ability to easily modify the behavior of an object.

Our approach was to begin with the actor formalism developed at the M.I.T. Artificial Intelligence Lab (Baker (1978), Lieberman (1981)), and adapt if for use as a programming system. The actor formalism has the following properties, which are ideally suited to our objective.

1. Instances of actor types are the executing and communicating objects in the system. Their behavior is specified by a "script" and they have memory which persists between invocations.
2. An actor communicates with other actors, known as it's "acquaintances", via messages. It is transparent to an actor sending or receiving a message whether the other actor is at the same node or a different node in a network of computers.
3. Actors execute asynchronously, upon receipt of a message.
4. Actors are highly modular with well defined interfaces. One cannot look inside an actor. One only knows its behavior in terms of the types of messages it accepts and the way it responds to those messages.
5. Every object is an actor; in particular, the messages are actors.

The actor-based programming system intentionally deviates from a pure actor formalism in the granularity of the actor objects, and in the way in which they are defined. In a pure actor system absolutely everything is an actor, actors are defined in terms of other actors, i.e. there are no numbers or arrays. In the actor-based programming system the objects implementing an application behave as actors. However, the internal specification of an

* Current address: Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

actor is PL/I-like and contains PL/I-like data types. While an actor specification can be as small as two lines it can legitimately grow to be hundreds of lines.

Our break from a pure actor formalism is basically pragmatic, but is vital to the success of the actor formalism as a programming discipline. At its origins, actors were presented as a new model of computation, not as a new programming language (cf. Baker(1978)). Users were expected to program in LISP, and use a compiler to generate computations in terms of actors. Our objective is a system in which users can directly program the actors participating in an application.

In addition, a pure actor system implies an overwhelming amount of parallelism and communication. While this is appropriate for an array of microprocessors (cf. Hewitt(1980)), we do not foresee this kind of processing configuration becoming readily available in the near future. Instead, we expect a local network of conventional single processor computers. The cost of a message transmission in this environment will be large enough to preclude the level of message communication in a pure actor system. By defining fewer larger actor objects one can perform more computation within each actor and reduce the degree of communication between actors. This makes it possible to achieve a match between the amount of parallelism and communication in the system and the capabilities of the available processors.

There are other object oriented programming disciplines which have properties similar to actors. In particular, SIMULA (Birtwistle, et al. (1973)), Smalltalk (Ingalls (1978)), and "abstract data type" languages like CLU (Liskov, et al. (1977)). A thorough comparison of these programming languages would be of interest, but would take a considerable amount of effort, and is outside the intent of this paper. We adopted the actor formalism because it most closely matched SBA's requirements, it provided a sound theoretical basis to build upon, and it promised to yield the most compact and efficient implementation of the programming system that we sought.

Section 2 introduces an example which will be referred to throughout the paper. It will be used to demonstrate both how actors are defined, and how applications can be implemented as a collection of communicating actor objects. Section 3 describes the basic constructs in the actor language, and completes the definition of the major actors in the example application. Section 4 describes additional features not needed in the example. Section 5 deals with the implementation, in particular, those aspects of the

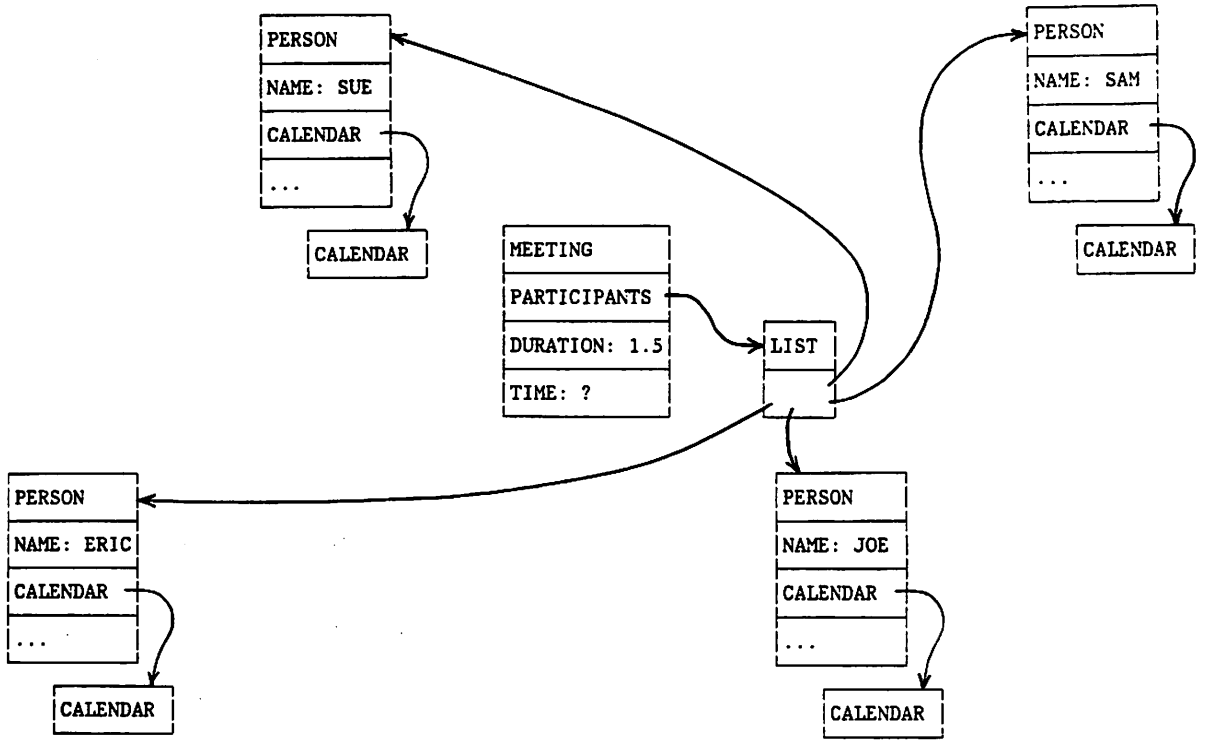


Figure 1. Scheduling a Meeting.

implementation concerning performance. Section 6 summarizes applications built with the current implementation. Section 7 summarizes the paper and describes the current status of the project.

2. An Example.

As an example of a business application implemented as a collection of actors, we consider the process of scheduling a meeting. A MEETING can be partially characterized by a list of its PARTICIPANTS, its DURATION and the TIME of the meeting. (We ignore other aspects of meetings such as topic, location, etc.) A meeting can be scheduled by inspecting the CALENDAR of each PERSON on the participant list, and scheduling the meeting for the earliest time period of the required duration which all calendars indicate to be currently unscheduled.

We will implement the scheduling process with the following actor types: MEETING, PERSON, CALENDAR, TIME_INTERVAL, TIME_INTERVAL_LIST, LIST, FIRST, and NEXT. Figure 1 depicts the major actors involved, with the arrows emanating from each actor representing its "acquaintances"--other actors known to it, and with which it can communicate. The scheduling process could proceed as follows:

1. A MEETING actor is created with a LIST of PERSON actors as its PARTICIPANTS acquaintance. The duration field is initialized.
2. The MEETING actor cycles over the PERSON actors on its LIST of PARTICIPANTS. It (asynchronously) broadcasts itself to all of the CALENDAR actors belonging to those PERSONs.
3. The CALENDAR actors, upon receipt of an unscheduled MEETING, create and return a TIME_INTERVAL_LIST containing the TIME_INTERVALs which a) are currently free, and b) exceed the DURATION required for the MEETING.
4. When a returned TIME_INTERVAL_LIST is received, the MEETING actor sends it to the POSSIBILITIES acquaintance, which is also a TIME_INTERVAL_LIST. POSSIBILITIES updates it's state to the intersection of its current state with the state of the TIME_INTERVAL_LIST message.
5. When all CALENDAR actors have responded, the MEETING sets its TIME field to be the TIME of the first TIME_INTERVAL in its POSSIBILITIES acquaintance. The newly scheduled MEETING actor again broadcasts itself to the original set of CALENDARs for inscription.

This solution to the scheduling problem exploits many of the properties of actors we feel to be useful. The PERSON and CALENDAR actors can be resident on separate personal computers, and distribution is obtained by merely sending messages to those actors. When MEETING broadcasts itself to the CALENDARs it initiates concurrent execution of the CALENDARs, thus taking advantage of natural parallelism in the application.

3. Actors.

Defining Actor Types.

Actor types are defined using the actor language, which is built on a subset of PL/I. An actor type definition has the following structure:

1. An ACTOR statement, giving the name and attributes of the actor type.
2. Zero or more INSTORE statements declaring instance variables.
3. Zero or more DECLARE statements declaring automatic variables.
4. Zero or more ACCEPT statements defining the actor's responses to messages.
5. Subroutines which can be invoked from within the ACCEPT clauses.
6. An END statement.

Figure 2 shows the overall structure of the definition of the MEETING actor described in the preceding example.

```
MEETING: ACTOR SERIAL;
  INSTORE
    PARTICIPANTS ACQ,
    DURATION FIXED,
    TIME FIXED,
    COUNT FIXED,
    POSSIBILITIES ACQ;
  DECLARE
    P ACQ;
ACCEPT('CREATE')
  ...
ACCEPT('TIME_INTERVAL_LIST')
  ...
END;
```

Figure 2. Outline of MEETING actor script.

One of the attributes which can be specified in the ACTOR statement is SERIAL. When specified, it forces the messages sent to an instance of the actor to be processed sequentially. Specifying NONSERIAL allows concurrent invocations of an actor.

The data types supported for automatic and instance variables are the PL/I data types plus the ACQUAINTANCE (ACQ) data type. An acquaintance variable can

contain the name of an actor. Having the name of an actor allows reference to that actor in a message transmission. The internal form of the name is not revealed to the programmer.

The INSTORE variables define the contents of **instance storage** for actors of the type being defined. The instance storage of an actor is a piece of storage which identifies the actor type, and contains variables whose values represent the current state of the instance. This state persists from one invocation of the actor to the next. Instance storage, therefore, is the permanent representative of the actor instance within the computing system. When an actor serves as the message in a transmission, it is the instance storage which is actually transmitted.

Each ACCEPT statement identifies a message type to be accepted, and the code to be executed upon receipt of a message of the specified type. If the code for an accept clause comprises more than one statement, it is enclosed in a DO-END block.

The source language specification for an actor type is compiled. This compilation process produces an executable **script** file and a template for the instance storage. Instance storage for actor's of type MEETING would be formatted as shown in Figure 3.

0	SCRIPT_ACQ
4	PARTICIPANTS
8	POSSIBILITIES
12	0
16	DURATION
20	TIME
24	COUNT

Figure 3. Instance Storage for MEETING Actors.

The SCRIPT_ACQ field is a required acquaintance variable which identifies a system defined SCRIPT actor. All instances of a given type point to a common SCRIPT actor which identifies the name of the type and the location of the executable script. Instance storage is allocated from a heap, and a garbage collector is used to reclaim the instance storage of actors which can no longer be accessed (i.e. those which no other actors have as acquaintances). Acquaintance variables are separated from the other variables in instance storage to facilitate garbage collection.

Creating Actor Instances.

An instance of an actor is created explicitly with a CREATE operation.

```
CREATE(type,parameter)
```

CREATE causes instance storage for an actor of type "type" to be allocated. If the script for that type contains an ACCEPT('CREATE') clause, the script is invoked and the create clause executed. The create clause specifies processing to be performed at the time of creation. The "parameter" in the CREATE operation is an actor which can be referenced in the create clause. In the absence of an ACCEPT('CREATE') clause, the instance storage for an actor is created with uninitialized instance variables. CREATE is a function which returns the new actor as its value.

Actor Communication.

An actor communicates with it's acquaintances by sending them messages. **Send** operations are provided for initiating **message transmissions.** A message transmission has four components which are set by the send operations. The first two are the **target,** which is the actor to receive the message, and the **message** actor itself. A third component, the **customer,** is the actor which should receive the reply message if one is generated as a result of the message transmission. The customer is frequently, but not necessarily always, the actor initiating the send operation. The last component in a message transmission, called the **client,** is a system defined actor used for synchronization, as will be described later. Note that all four components of a message transmission are necessarily acquaintances of the actor performing the send operation (the "sender"). The sender itself is not automatically a component of the resulting message transmission, although it may function as the target, message or customer.

Asynchronous communication is provided by a SENDC (Send-Continue) operation,

```
SENDC(target,message,customer,client).
```

The SENDC operation provides a "fork" capability, asynchronously sending the message to the target. Control returns immediately to the sender (i.e., the message will not necessarily have been processed), and the target and sender execute concurrently. In a SENDC operation one can specify all four components of the message transmission. If either the customer or client is not specified it defaults to the current customer or client, respectively, of the invoking actor.

The system provides three send operations for synchronous message communication. For all three, the sender specifies only the target and message components of the message transmission. The customer component is set according to the type of send operation performed. The client component is always set to the client of the sender.

```
SENDR(target,message)
```

A SENDR (Send-Reply) operation, provides a synchronous transfer of control similar to the procedural call-return mechanism. The sender serves as the customer, and waits for a reply message. The SENDR operation returns an acquaintance identifying a reply message.

```
SEND(target,message)
```

A SEND operation, is identical to SENDR except it does not expect a return message, and if one is generated, it is ignored.

```
SENDE(target,message)
```

A SENDE (Send-End) operation, differs from the others in that the invocation of the sender is terminated. The sender's current customer serves as the customer for the message transmission, rather than the sender itself, as in the other two synchronous send operations. As a result, any reply message generated by this transmission will be returned directly to this customer. A SENDE operation is functionally equivalent to

```
RETURN(SENDR(target,message))
```

but avoids an unnecessary change of contexts during the return processing. It is a convenient means of forwarding a message to another actor for processing.

Between any pair of actors, messages are guaranteed to be processed in the order in which they are sent. For any one actor the messages are processed in the order of arrival.

Invocation of Actors.

Every message transmission causes a new invocation of the target actor. If the target's instance storage is currently residing on secondary storage, it is read into main memory. If the script for actors of the target's type is not resident, it is dynamically loaded. Automatic storage is provided on a stack, and control is transferred to the actor's script. The script executes the accept clause corresponding to the type of the message received.

Special acquaintance values may be accessed during execution. These values are constant during any given invocation of an actor, although they may change from one invocation to the next. These special values are:

SELF - the name of the executing actor instance.

MESSAGE - the name of the actor whose receipt, as a message, caused the current invocation. In an ACCEPT clause for a "CREATE", MESSAGE is the name of a parameter actor.

CUSTOMER - the name of the actor which will receive any reply from the current invocation.

CLIENT - the client for the current invocation.

NULL - the empty actor reference.

NOTHING - the value which, when used in a RETURN statement, will suppress the sending of a reply to the customer.

When the target actor completes processing of the received message it returns either an actor acquaintance identifying an actor which constitutes the return message, a NULL acquaintance, or NOTHING.

The processing performed by an actor in response to the messages it accepts is independent of the manner in which they are sent. Its processing is the same whether messages are sent via SENDR, SEND, SENDE or SENDC operations. The actor which sends a message, however, must be aware of the kind of response generated by the target actor when it receives messages of the type being sent. In particular, it is an error to send a message via a SENDR operation if the processing of that message will not eventually produce a valid reply message.

Set and Get References.

The code in an actor's script may refer to the instance variables of another actor by using the following syntax.

```
acquaintance->type-name.instance-variable-name
```

The meaning of such a reference depends upon where it appears in a statement. If it is the target of an assignment, the meaning is that a new value is to be placed in the named actor's instance storage. This is called a set reference. When the expression appears in another context, it is a get reference, and means that the current value of the variable in the named actor's instance storage replaces the expression in that context.

The use of set and get references appears to violate the rule that actors may only inspect and modify their own instance storage. However, the violation is only apparent. These expressions are compiled into object code which (when necessary) creates and sends one of a special set of message actors to the actor which owns the instance storage. These messages are processed by accept clauses which are implicitly included in the actor's script because of its instance variables. These implicit accept clauses simply update or return the value of an instance variable. Thus, the actor itself performs the requested modification or returns the requested value. The parenthetical reference is to the fact that the object code may directly access the actor's instance storage when the results of a direct reference are indistinguishable from those obtainable by sending a message. Notice that this system does not inherently provide the benefits of "information hiding", as found in the abstract data type languages. Users who wish to approximate those benefits may replace the implicit

```
MEETING: ACTOR SERIAL;
 INSTORE
  PARTICIPANTS ACQ,
  DURATION FIXED,
  TIME FIXED,
  COUNT FIXED,
  POSSIBILITIES ACQ;
 DECLARE
  P ACQ;
ACCEPT('CREATE')
 DO;
  (Initialize PARTICIPANTS and DURATION.
   Set POSSIBILITIES to a TIME_INTERVAL_LIST
    encompassing all of the future.
   Initialize TIME and COUNT to zero.)
  DO P = SENDR(PARTICIPANTS,CREATE('FIRST'))
     REPEAT SENDR(PARTICIPANTS,CREATE('NEXT'))
     WHILE(P ¬= NULL);
   SENDC(P->PERSON.CALENDAR,SELF);
   COUNT = COUNT + 1;
  END;
 END;
ACCEPT('TIME_INTERVAL_LIST')
 DO;
  SEND(POSSIBILITIES,MESSAGE);
  COUNT = COUNT - 1;
  IF COUNT = 0 THEN
   DO;
    TIME = SENDR(POSSIBILITIES,CREATE('FIRST'))
             ->TIME_INTERVAL.TIME;
    DO P = SENDR(PARTICIPANTS,CREATE('FIRST'))
      REPEAT SENDR(PARTICIPANTS,CREATE('NEXT'))
      WHILE(P ¬= NULL);
     SENDC(P->PERSON.CALENDAR,SELF);
    END;
   END;
 END;
END;
```

Figure 4. Specification for MEETING actors.

accept clauses for the special set and get messages with clauses which explicitly filter or prohibit access to instance variables.

Completion of the example.

We now return to the problem of scheduling a meeting. With the facilities described so far, we can show how some of the actors constituting the solution might be implemented.

First, Figure 4 completes the definition of the MEETING actor which was outlined in Figure 2. The overall functioning of this actor has been described in section 2, and will not be repeated here. Notice, however, MEETING's use of the SENDC operation to asynchronously broadcast itself to the CALENDARs of all PERSONs on its list of PARTICIPANTS. The first broadcast occurs in the ACCEPT('CREATE') clause to request available time intervals. The second one announces the time of the meeting after receipt of the final time interval list.

Figure 5 shows the actors involved in the operation of time interval lists. Such a list consists of a main TIME_INTERVAL_LIST actor plus a singly-linked chain of ELEMENT actors. TIME_INTERVAL_LIST actors must have the SERIAL attribute because the instance variables, HEAD and CURRENT, can be modified during an invocation, and there is no guarantee that multiple users won't simultaneously send to a TIME_INTERVAL_LIST. (Note, in contrast, that an ELEMENT actor is only known to the actor which created it. Because of the pattern of messages sent to ELEMENT actors, it is clear that they do not need to be SERIAL, even though they, too, modify their instance storage.)

In the ACCEPT('CREATE') clause, a TIME_INTERVAL_LIST initializes its instance storage, and returns its own name to its customer. Upon receipt of a TIME_INTERVAL message, it creates and chains a new ELEMENT actor. (The chaining process is assisted by the "CREATE" code in the ELEMENT script.) It then sets the VALUE acquaintance of the new ELEMENT actor to the name of the new TIME_INTERVAL with the statement:

```
HEAD->ELEMENT.VALUE=MESSAGE;
```

A FIRST message causes a TIME_INTERVAL_LIST to position its CURRENT acquaintance at the HEAD of its list of ELEMENTs, and then to send itself a NEXT message. A NEXT message causes it to update its CURRENT acquaintance, and then to return the result of a get operation for the VALUE field of its old CURRENT ELEMENT with the statement:

```
RETURN(TEMP->ELEMENT.VALUE);
```

4. Additional Facilities.

Actor Names.

Users of the system only know that an acquaintance variable contains the "name" of an actor with which it can communicate. The internal form of the name is not revealed. The implementation, however, recognizes three different kinds of names, **local names, global names,** and **external names.** The local name of an actor is simply the address of it's instance storage. It provides direct access to an actor, but only exists while an actor is resident in main memory. Local names are not permanently assigned, and if an actor moves from one node to another or to secondary storage and back to main memory, it will at different times have different local names. A global name

```
TIME_INTERVAL_LIST: ACTOR SERIAL;
 INSTORE
  HEAD        ACQ,
  CURRENT     ACQ;
 DECLARE
  TEMP        ACQ;
ACCEPT('CREATE')
 DO;
  HEAD,CURRENT = NULL;
  RETURN(SELF);
 END;
ACCEPT('TIME_INTERVAL')
 DO;
  HEAD = CREATE('ELEMENT',HEAD);
  HEAD->ELEMENT.VALUE = MESSAGE;
 END;
ACCEPT('FIRST')
 DO;
  CURRENT = HEAD;
  SENDE(SELF,CREATE('NEXT'));
 END;
ACCEPT('NEXT')
 DO;
  IF CURRENT = NULL THEN RETURN(NULL);
  TEMP = CURRENT;
  CURRENT = CURRENT->ELEMENT.FPTR;
  RETURN(TEMP->ELEMENT.VALUE);
 END;
ACCEPT('TIME_INTERVAL_LIST')
 DO;
  (Modify current state by intersecting
   with the state of the message.)
 END;
END;
```

```
ELEMENT: ACTOR;
 INSTORE
  FPTR     ACQ,
  VALUE    ACQ;
ACCEPT('CREATE');
 DO;
  FPTR = MESSAGE;
  RETURN(SELF);
 END;
END;


TIME_INTERVAL: ACTOR;
 INSTORE
  TIME FIXED,
  DURATION FIXED;
END;


FIRST: ACTOR;
END;


NEXT: ACTOR;
END;
```

Figure 5. Additional actor specifications for meeting problem.

identifies an actor at another node. It is implemented as the local name of a system defined actor, of type GLOBAL, which contains the remote node's identifier and the local name of the actor within that node. An external name acquaintance value may be used to identify an actor which has been assigned an external name (a character string). It is implemented as the local name of a system defined actor, of type EXTERNAL, which contains the actual character string name. An actor identified by it's external name may be resident in main memory, on secondary storage, or at another node.

When an actor is created it only has a **local name.** It can be assigned an external name by means of a TITLE operation.

```
TITLE(acquaintance,external name)
```

where "acquaintance" identifies the actor instance to be named, and "external name" is a character string of the form "type-name.instance-name". An external name must be unique within each node. The system does not enforce uniqueness of external names throughout the whole network, and use of an external name will always result in a reference to a local actor with that name before it results in a reference to an actor at a remote node with the same name.

The external name of an actor can be used in scripts as an **acquaintance constant** which references the actor. For example, the phrase

```
'PERSON.ERIC'->PERSON.CALENDAR
```

can be used as a set or get reference to the CALENDAR instance variable of the actor instance ERIC of type PERSON. An acquaintance constant may appear anywhere in a script that an acquaintance variable is allowed, except as the target of an assignment operation.

Storing Actors.

Actors which have been assigned an external name may be saved on secondary storage with a STORE operation,

```
STORE(acquaintance)
```

Stored actors persist across session boundaries, and exist until explicitly destroyed. A stored actor is accessible via it's external name, and will be automatically read into main memory and assigned a local name when referenced.

It is not sufficient to store just the specified actor. Take for instance a named TIME_INTERVAL_LIST actor which uses a linked list of unnamed ELEMENT actors to identify the elements of the list. One could not store just the list actor, but would have to also include the list of element actors. Because of this, a STORE operation results in storing the following collection of actors.

1. The specified named actor.
2. All unnamed acquaintances accessible from that actor without an intervening named actor.
3. For each accessible named actor, a system EXTERNAL actor containing the actor's external name.

The collection of actors is stored in the form of a single variable length record called a **stored actor record.**

Remote Actor Invocations.

If the target of a message transmission identifies an actor at a remote node, then a system REMOTE actor is invoked in place of the target. The REMOTE actor interfaces with the network to cause the remote invocation of the target actor. The message actor is either moved to the remote node or a GLOBAL actor identifying the message is sent in its place. At the remote node the GLOBAL actor is interpreted as a global name acquaintance.

When an actor is moved to another node, it is done by forming a **network actor record.** A network actor record is the same as a stored actor record, except that the originating actor need not be named. Additionally, the acquaintances identified by local names, can either be included in the record or be represented by system GLOBAL actors. A network actor record is a portable form of an actor which can be sent around the network. Upon arrival at a node it is read into memory by the same mechanism used to read stored actor records.

Synchronization.

Several mechanisms are provided for synchronizing actor execution. Together, these mechanisms constitute a general architecture within which any desired synchronization control structure can be built.

At the lowest level, actors can be marked **non-serial** or **serial.** A message transmission to a non-serial actor can cause the invocation of that actor to occur immediately, even when the actor is already executing (with a message from a separate transmission). In order to support such concurrent processing, the code in all actor scripts is reentrant. Transmissions to non-serial actors are never queued. On the other hand, a transmission to a busy serial actor is placed on a FIFO queue of transmissions associated with that actor. This queue supports the guarantee, mentioned earlier, that messages will be processed by an actor in the order in which they are received.

The second level of synchronization is the **client.** Clients provide the "join" that corresponds to the "fork" capability given by send-continue (SENDC). CLIENT is a system actor type. When a non-NULL client actor is specified for a transmission started by a SENDC operation, it keeps track of the asynchronous execution path started by that SENDC. In particular, it keeps a count of such execution paths, incrementing the count for each new SENDC with that client, and decrementing the count when each path terminates. When the count reaches zero--i.e., all asynchronous execution paths begun with that client have ended--the client actor initiates a transmission with target and message actors which were specified at the creation of the client.

```
  ...
  MYCLIENT = CREATE_CLIENT(SELF,
                    CREATE('RESUME'));
  SENDC(ACTOR1,MESSAGE1,,MYCLIENT);
  SENDC(ACTOR2,MESSAGE2,,MYCLIENT);
  RETURN(NOTHING);
  ...
ACCEPT('RESUME')
  ...
```

Figure 6. Using Clients for Synchronization.

In the example in Figure 6, MYCLIENT is a client actor whose target acquaintance contains the name of the actor which created it (because of the reference to SELF), and whose message acquaintance is an actor of type RESUME. Before control is relinquished by the RETURN statement, the two SENDC operations will have caused the client's count to be set at 2. When the processing initiated by MESSAGE1 and MESSAGE2 has been completed, the client's count will have been decremented to zero. The client will then send the RESUME actor as a message to the actor which created it. Notice, finally, that the target actor has an accept clause for messages of type RESUME; this is the point at which the "join" operation occurs.

Activities are provided at the third level of synchronization. They are similar to clients in that they can monitor the progress of asynchronous execution paths, and they allow the specification of a transmission which is to occur at the termination of those paths. In addition, activities may be cancelled and are capable of being hierarchically nested within one another. If an actor executing as part of one activity issues a SENDC operation with another activity specified as client, then the transmission issued at the

completion of the child activity is considered to be part of the parent activity. Parent activities may execute concurrently with their children, but they are never considered complete until all of their children have either terminated or been cancelled.

```
  ...
  ACTIVITY1 = CREATE_ACTIVITY(SELF,
                    CREATE('RESUME1'));
  ACTIVITY2 = CREATE_ACTIVITY(SELF,
                    CREATE('RESUME2'));
  SENDC(ACTOR1,MESSAGE1,,ACTIVITY1);
  SENDC(ACTOR2,MESSAGE2,,ACTIVITY2);
  ...
ACCEPT('RESUME1')
 DO;
  CANCEL(ACTIVITY2);
  ...
 END;
ACCEPT('RESUME2')
 DO;
  CANCEL(ACTIVITY1);
  ...
 END;
 ...
```

Figure 7. Using Activities for Synchronization.

An example of the use to which the ability to cancel activities may be put appears in Figure 7. When either activity completes, the first action of the "join" code is to cancel the other activity. (Note that the join for the cancelled activity will never be executed.) If ACTOR1 and ACTOR2 were, for example, trying to produce the same result using different algorithms, this type of control structure would allow the cancellation of the losing algorithm as soon as the other one succeeds.

The activity mechanism is entirely implemented with the system actor type, ACTIVITY. The script for activity actors is illustrated in Figure 8. The instance variables of activity actors include a fixed point child count, as well as target, message, and parent activity. The child count and the parent activity acquaintance form the basis for the ability to nest activities. Every new activity becomes a child of the activity under which it is created. Part of the creation process for activities involves sending a YOU__HAVE__A__CHILD message to the parent, causing it to increment its child count. When an activity and all of its children have finished, it transmits its message acquaintance to the target, and sends a YOU__LOST__A__CHILD message to its parent. The parent, in turn, decrements its child count.

When created, each activity actor creates a client actor to manage its own actor invocations (as opposed to those of its children). Upon completion of these invocations, the client sends a YOUR__WORK__IS__DONE message to the activity actor which created it. When this happens, the activity will terminate as soon as it detects that its child count has reached zero.

```
ACTIVITY: ACTOR SERIAL;
 INSTORE
  TARGET ACQ,
  MSG ACQ,
  ORIGINATOR ACQ,
  PARENT_ACTIVITY ACQ,
  MY_CLIENT ACQ,
  MY_WORK_IS_DONE BIT(1),
  CHILD_COUNT FIXED;
ACCEPT('CREATE')
 DO;
  (Initialize the TARGET and MSG acquaintances.
  Set PARENT_ACTIVITY to the name of the current
  activity, or NULL if there is none.)
  IF PARENT_ACTIVITY ¬= NULL THEN
    SEND(PARENT_ACTIVITY,
         CREATE('YOU_HAVE_A_CHILD'));
  ORIGINATOR = CUSTOMER;
  MY_WORK_IS_DONE = '0'B;
  CHILD_COUNT = 0;
  MY_CLIENT = CREATE_CLIENT(SELF,
            CREATE('YOUR_WORK_IS_DONE'));
  RETURN(MY_CLIENT);
 END;
ACCEPT('YOU_HAVE_A_CHILD')
 CHILD_COUNT = CHILD_COUNT+1;
ACCEPT('YOU_LOST_A_CHILD')
 DO;
  CHILD_COUNT = CHILD_COUNT-1;
  IF CHILD_COUNT = 0 & MY_WORK_IS_DONE THEN
   DO;
    IF PARENT_ACTIVITY ¬= NULL THEN
      SEND(PARENT_ACTIVITY,
           CREATE('YOU_LOST_A_CHILD'));
    SENDC(TARGET,MSG,ORIGINATOR,NULL);
   END;
 END;
ACCEPT('YOUR_WORK_IS_DONE')
 DO;
  MY_WORK_IS_DONE = '1'B;
  IF CHILD_COUNT = 0 THEN
   DO;
    IF PARENT_ACTIVITY ¬= NULL THEN
      SEND(PARENT_ACTIVITY,
           CREATE('YOU_LOST_A_CHILD'));
    SENDC(TARGET,MSG,ORIGINATOR,NULL);
   END;
 END;
END;
```

Figure 8. Definition of the ACTIVITY actor.

5. Performance Considerations.

It must be possible to develop actor-based applications which meet acceptable levels of performance, i.e., performance comparable to a direct implementation in a high

level language. The following summarizes those aspects of the actor programming system directed at performance.

1. Actor scripts are compiled, rather than interpreted. Compiling will generally yield superior performance. The ability to define larger actors, of a size comparable to a procedure, allows one to capitalize fully on compiler optimization techniques.

2. Messages are passed by reference, and are copied only when it is necessary for transmission across a network. This is particularly important in an actor system where messages are actors and can be large. Actors, such as the MEETING actor in the example, frequently pass themselves to an acquaintance as a message, even though a much smaller message could have sufficed.

3. Symbolic references to actors (external names) are resolved to direct memory addresses. This avoids having to resolve symbolic names on every reference. In addition, a hash table is used to provide quick resolution of external names of actors already resident in main memory.

4. An executing actor-based application will typically create a large number of actors. Thus, actor instances must be inexpensive to create and have a minimal amount of space overhead. The processing cost of creating an actor is essentially the cost of allocating a structure in an area. The storage costs for an actor can be as small as 8 bytes, a 4 byte header and 4 bytes for the obligatory script acquaintance.

5. The large number of send operations which will be performed in a typical application make the cost of a send operation a significant factor in overall performance. Some send operations will be unavoidably expensive, requiring, allocation of a stack, resolving external names, network transmissions, dynamic loading of an actor's instance storage from secondary storage, and dynamic loading of a script. The majority of send operations, however, will be synchronous (SEND, SENDR, SENDE) transmissions where both the target actor's instance storage and script are already resident in main memory. The system design strives to make these transmissions as efficient as a normal procedure call. When the send mechanism is invoked the target acquaintance is the address of the target's instance storage, and the first acquaintance in that instance storage block points to a SCRIPT actor containing the address of the script to be executed. The send mechanism can simply transfer control to the identified script. The script runs on the current stack, i.e. the stack of the sending actor invocation. For SEND and SENDR operations it executes immediately above the sender, and for SENDE operations in replaces the sender (the sender's invocation is terminated). The net effect is that the target actor invocation occurs just above the customer actor invocation, as if the target had been called by the customer. This allows a message to be returned to the customer via a normal procedural return statement.

6. The process of storing an actor on secondary storage, and subsequently reading it back into main memory is a potential performance problem. The process involves a potentially large collection of actors. The format of the actors within a stored actor record is kept the same as in main memory, with acquaintances converted to offsets from the beginning of the record. This makes it possible, when reading a stored actor, to process the entire record as a unit, and avoid separate allocations for each component actor. The stored actor record is read directly into a block of storage within the area used for instance storage. A single pass is made over the the record to convert the acquaintance record offsets to local names. In addition, SCRIPT actors are not stored. Instead, their names are placed at the end of the stored actor record, and they are recreated when the record in read. External name acquaintances are intentionally not resolved when a stored actor is read. The send mechanism dynamically resolves the external name acquaintances upon first use, thus reestablishing connections to named actors which were severed during the storing process. A beneficial by-product of storing and reading an actor is that the actor and its unnamed acquaintances become physically adjacent in main memory.

6. Applications.

Part of the motivation for building the actor system is to explore the usefulness of the actor model of programming for building substantial applications. This exploration naturally involves the programming of actual applications.

An application is a subsystem of actors communicating with each other. The application designer must approach his task differently than if he were using a traditional programming system. For example, he cannot use global data structures, since actors provide none. He must decompose his problem into active entities--which he then implements as actors--rather than passive data structures operated upon by arbitrary control structures. The interfaces among these entities must be well-defined. Distributed applications become easier to define, since distribution is built into the actor model. Asynchrony is much more accessible than in other systems, and the designer should

incorporate it in intelligent ways. As the designer gains experience with the actor model, he is rewarded with increased ease in developing and implementing system structures for his applications.

We summarize some of the applications built, and being built, using actors.

Our first application was an implementation of the OMEGA knowledge representation system (Hewitt, Attardi, and Simi (1980)). OMEGA represents knowledge as a network of "descriptions". At the lowest level, descriptions represent the objects of the world being modelled, and the links in the network represent inheritance and attribution relationships among those objects. The more interesting aspect of OMEGA is that descriptions may also represent axioms which further describe the semantics of the world being modelled.

The implemented system provides a language in which OMEGA statements and requests can be entered. Statements are transformed into their corresponding descriptions and merged with the network of existing descriptions. Requests explore the inheritance and attribution links, under the control of the axioms in existence, and yield information to the user. Some of the description types supported, along with examples of the description language, are given below:

Atomic - `John`
Instance - `(a child)`
With-attribution - `(with sex male)`
Inheritance - `(is John (a child (with sex male)))`
Variable - `=x`
Implication - `(=> (is =x (a child (with sex male))) (is =x (a boy)))`

The implementation represents each description type as an actor type; actor instances, therefore, represent individual descriptions. The acquaintances of a description actor include the other descriptions linked to it in the semantic network. Additional actors provide such functions as user communication, input parsing, output formatting, and triggering of user-supplied axioms. Altogether, about forty actor types, with scripts ranging in size from two lines to four-hundred lines of actor code, were required for the implementation. During an OMEGA session, thousands of actor instances may be in existence at one time.

A second actor application is an end-user window interface to the SBA system. The interface gives the user simultaneous access to multiple overlapping windows on his terminal screen. All windows are active at all times; this contrasts with systems - like Smalltalk - in which the user interacts with a single "active" window, and in which activation of a new window forces a "focus shift" on the part of the user. Individual SBA windows are controlled by instances of a WINDOW actor type. Each one can change its own name, size, color, border, and height with respect to other windows. A window can also initiate office applications, manipulate multiple graphical images belonging to the applications, and scroll over these images, all in response to user commands. The window actors communicate with an actor of type SCREEN by sending it PICTURE messages. The screen actor manipulates the multiple pictures of windows, creating, moving, and overlapping them on command; it also notifies the window actors of user activity at the terminal, by sending modified PICTURE actors.

The SBA window system provides a unified application development and execution environment at an office workstation. Additional applications, also built of actors, operate within the window environment. These applications are "graphical" in the sense that they communicate with WINDOW actors about their status by sending and receiving PICTURE messages. They include:

- SBA tables and files which support Query-by-Example operations, and whose underlying relational access method was simulated by an OMEGA network,
- SBA forms,
- a personal calendar,
- a continuously running digital clock, and
- a lightpen-operated pocket calculator.

We must not forget that one of the more important applications of actors to date has been the implementation of the actor system itself. The use of actors to implement clients, activities, scripts, directories, loaders, and names, testifies to the generality of the actor programming model.

7. Summary and Status.

The actor-based programming system takes the formal theory of actors and adapts it for use as a language and system for developing applications. Whereas actor systems have previously been used only in artificial intelligence research, they can now be employed in other areas. In particular, we propose to use ours for programming distributed business applications. An application is implemented as a collection of independently executing actors communicating by messages. They can be distributed over a network of computing nodes, and permanently stored on disk. A naming architecture allows actors to communicate in a uniform way regardless of their location or status.

Actor types are defined through a PL/I-like language which specifies both a storage area to be associated with

the actor and its behavior in response to the messages it accepts. This behavior may involve changing the state of its instance storage area, creating other actor instances, initiating communication with its actor acquaintances, and returning a message to the actor which caused it to be invoked. Mechanisms are provided for synchronizing concurrently executing actors.

While the system retains the desirable properties of actors, performance has been a primary concern, and we have been careful to avoid the inefficiencies usually found in object-oriented message-passing systems. The size of an actor specification can vary from two lines to hundreds of lines of PL/I-like code, and is compiled with an optimizing compiler. Messages are passed by reference, and are only copied when transported across the network. Names of acquaintance actors are dynamically bound to direct memory references upon first use.

Status.

The prototype processor for the actor language was built using an existing optimizing compiler for a subset of the PL/I language together with the standard PL/I macro preprocessor (I.B.M. (1976)). Syntactic and semantic limitations imposed by the macro preprocessor resulted in the implemented language differing somewhat from the descriptions given in this paper (see Byrd (1980)). Continued development of the actor system should include the implementation of a compiler for the actor language.

Some of the described language features, such as the ability to cancel activities, have not yet been implemented. The concept of inheritance, which is similar the Smalltalk notion of **superclass**, is needed, as well as a mechanism by which an actor can delegate the handling of a message to another actor. The information hiding properties of the abstract data type languages, such as CLU, should be made available. The serial/non-serial attribute could be made a function of the message type being processed, and an actor's instance storage could be allowed to dynamically vary in size.

The existing prototype implementation is for a single actor node in a VM/370 virtual machine. It is being extended to multiple actor nodes residing in a network of virtual machines. This will simulate the intended processing configuration of a local network of small computers, and allow experimentation with distributed actor applications. This will be a vehicle for conducting research on the difficult problems confronting distributed applications, including deadlock detection/prevention, crash recovery, and distributed garbage collection. An actor-based system should provide the potential for unique and interesting solutions to problems in these areas.

Acknowledgements.

We wish to thank Carl Hewitt for many long and exciting discussions of the actor formalism and the description system OMEGA, especially during his stay at the IBM T. J. Watson Research Center in the summer of 1980. We also thank John Lucassen for his energetic work during the early stages of the actor system implementation, and Ken Niebuhr, the other member of the SBA group, for his insight during many discussions.

References.

Baker, H. G., Jr. (1978) **Actor Systems For Real-Time Computation,** M.I.T. Ph.D. Thesis. Laboratory of Computer Science Technical Report MIT/LCS/TR-197.

Birtwistle, Dahl, Myhrhaug, and Nygaard (1973) **SIMULA Begin,** Auerbach.

Byrd, R. J. (1980) "Macros and Coding Conventions for SBA Boxes," SBA project memo.

de Jong, S. P. (1980) "The System for Business Automation (SBA): A Unified Application Development System," in **IFIPS Congress 80 Proceedings,** North Holland Publishing Company.

de Jong, S. P., and R. J. Byrd (1980) "Intelligent Forms Creation in the System for Business Automation (SBA)," I.B.M. Research Report RC 8529.

Hewitt, C. (1977) "Viewing Control Structures as Patterns of Passing Messages," in **Artificial Intelligence** Vol. 8, pp. 323-364.

Hewitt, C. (1980) "The Apiary Network Architecture for Knowledgeable Systems," M.I.T. AI Lab Memo.

Hewitt, C., G. Attardi, and M. Simi (1980) "Knowledge Embedding in the Description System OMEGA," in **Proceedings of the First National Annual Conference on Artificial Intelligence** American Association for Artificial Intelligence, August 1980.

I.B.M. (1976) **OS PL/I Checkout and Optimizing Compilers: Language Reference Manual,** I.B.M. form GC33-0009.

Ingalls, D. H. H. (1978) "The Smalltalk-76 Programming System: Design and Implementation," in **Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,** pp. 9-16.

Lieberman, H. (1981) "A Preview of Act 1," M.I.T. AI Lab Memo 625.

Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert (1977) "Abstraction Mechanisms in CLU," in **Communications of the ACM,** Vol. 20, no. 8, pp. 564-576.

Zloof, M. M., and S. P. de Jong (1977) "The System for Business Automation (SBA): Programming Language," in **Communications of the ACM,** Vol. 20, no. 6, pp. 267-280.

Supporting Organizational Problem Solving with a Workstation

Gerald Barber

Massachusetts Institute of Technology
Room 812
545 Technology Square
Cambridge, Mass. 02139
(617) 253-7891

EXTENDED ABSTRACT

This paper describes an approach to supporting work in the office. Using and extending ideas from the field of Artificial Intelligence (AI) we describe office work as a problem solving activity. A knowledge embedding language called Omega is used to embed knowledge of the organization into an office worker's workstation in order to support the office worker in his or her problem solving. A particular approach to reasoning about change and contradiction is discussed. This approach uses Omega's viewpoint mechanism.

1. Office Semantics

Office Semantics is the study of information intensive organizational work. Its name reflects the concern with *the intent behind the act.* Office Semantics is concerned with understanding the reasons behind the physical tasks that are performed in organizational work. To understand organizational behavior a distinction is made between the *application structure* of the organization and its *organizational structure.*

We make the distinction between the application and organizational structures because they are distinct bodies of knowledge that react to different forces of change and because of their differing functions in the organization. The organizational structure changes in response to forces such as work force mobility and change in the formal structure of the organization. The application structure responds to changes in laws governing aspects of the application, for example tax laws. Changes in product and service requirements also change the application structure. The organizational structure realizes the problem solving strategies necessary to fulfill the requirements of the application structure. The

distinction between the application and organization structures does not imply that one is more relevant to organizational work than the other. Organizational work must conform to the constraints and rules derived from both the organizational and application structures.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0079 $00.75

2. Office Work as Problem Solving

A fundamental premise is that problem solving is an inherit characteristic of office work. Office work has 4 fundamental characteristics shown below.

2.1. Open-Ended Knowledge World

In contrast to some knowledge worlds investigated by AI researches such as the Blocks World [6] the world of organizational knowledge is not a closed knowledge world. The complete set of actions relevant to the organizational world is unknown and unknowable. The set of all possible states are unknowable as are all possible alternatives for achieving a goal. The result is that unforeseen situations are a common occurrence. This is as much a property of the perceiver of the world as it is of the world itself since it is our assumption that the perceiver is of limited cognitive capabilities.

The open-ended character of the organizational knowledge world places demands on the kind of description system used to describe organizational knowledge. In particular the description system must be able to assimilate new information about actions, situations, and alternatives to achieving goals in an incremental fashion. The description system must be able to reason with partial information about problem solving states.

2.2. An Evolutionary Environment

Organizational environments are continuously changing. Any attempt to understand and describe organizational behavior must cope with the problem of trying to hit a moving target. This is a central problem both in talking about organizations and in doing work within organizations. A description system must be able to describe an organization that is continuously changing. A description system must also furnish tools to manage change so office workers may use it in performing their tasks. The Omega description system provides the Viewpoint mechanism to describe and reason about change.

2.3. Perception of Cognitive Processes from Overt Physical Actions

Trying to understand what task someone is doing and the reasons for each action performed in carrying out the task by watching the person perform the task is in general not possible. Information used in performing the task is not manifest in the physical actions the task entails. Even asking someone how they accomplish a particular cognitive task yields at best partial information and often apparently contradictory information. This limitation implies two constraints on office knowledge: first, the quality of information gathered by observation or interview is limited. Second, and hence, more effective

methodologies -are desirable. Partly because of this problem the approach we take is to *support* problem solving rather than replace the individuals in an organization doing the problem solving.

2.4. Describing Cognitive Processes

In order to discuss cognitive processes they must be describable. In the general case it is not possible to describe all mental activity involved when a person is thinking through a problem. Describing all mental activity involved is also not desirable since much of it may be irrelevant to the problem at hand and idiosyncratic to the particular individual. The goal of describing cognitive processes is not to develop a psychological theory of the individual in an organizational setting but to describe the individual in a way that--taken in aggregate--explains organizational behavior.

The premise is that there is a way to describe an *organizational person* in terms of application and organizational knowledge. In adopting this premise an assumption is made that an organization works in such a way as to factor out the individual idiosyncrasies of its members. The reason for making this assumption is that many organizations have similar behaviors but are made up of diverse personalities.

3. Describing Office Work

The purpose of describing office work is to make explicit the work that is done in the office. This includes the mental and physical activities that an office worker engages in and the reasons for these activities. An approach to characterizing work in the office is to consider it as organized in procedures in a fashion similar to the computer science notion of procedure. In this way office work would be described as a sequence of steps with decision points to manage flow of control.

3.1. Pitfalls of a Procedural Description Methodology

As described in [1], a procedural characterization is problematic for several reasons. Even routine tasks in offices are beset by a plethora of contingencies. In a procedural approach it is necessary to foresee the possible alternatives that may arise and write the procedure to accommodate them. When trying to describe office work in this step by step manner it becomes clear that all the alternatives cannot be enumerated. Determining what the alternatives are is part of what office work is; all alternatives cannot be determined in advance. As a result a procedural approach is not a very useful style of work description.

3.2. Explicit Representation of Goals and Actions

A description of office work in terms of goals and actions is a direct way of characterizing office work. A procedural description of an order entry task, for example, succinctly characterizes the important points of the task. But precisely because of its succinctness a procedural description suffers from two defects: first, it glosses over minor details that may be problematic or critical in practice; second, the reasons for the actions specified by a procedural description must be inferred. Thus if it is impossible to fulfill a requirement in the procedural description, such as obtain the delivery address for an order, the office worker must rely on intuition and experience to select an alternative action. The more desirable approach is to state explicitly the reasons the action is needed--the goals the action achieves.

The explicit representation of goals and actions provides a recourse to handle unexpected contingencies. Office workers are able to handle unexpected contingencies in their daily work because they know the goals of the office work and because they know what actions are needed to achieve the goals of the office work. These goals and constraints are often implicit in the work and in the office workers' knowledge of their work. If a particular action cannot be performed the computer system can possibly suggest an alternative action. Failing this the office worker can use the computer system to examine the goals an alternative action must inherit from the action that cannot be performed. Together, the office worker and computer system can construct a new plan of action that maintains the necessary constraints and makes progress toward achieving the goals in question.

4. Supporting Office Work

Omega provides a uniform framework within which to implement tools to support an office worker's problem solving. This has the benefit that different tools may cooperate easily in achieving the goals of particular office tasks. The benefits of using descriptions to represent knowledge about an organization.

Knowledge is embedded in the form of descriptions about objects in the system and the relationships between these objects. Some office systems have taken the stand that forms are the basic element of the system, an attempt is then made to represent everything in the system using forms. We view descriptions as the basic element of the system. Since the knowledgebase is represented using Omega's description lattice data does not have to be cast in a rigid form as it does in traditional data processing applications. The consequence is that office tasks may be reasoned about more on an individual basis.

5. Viewpoints

Viewpoints may be thought of as repositories for descriptions and thus statements. Viewpoints are reminiscent of McCarthy's situational calculus [3] and the contexts of QA4 [4] and Conniver [5]. The most notable difference between viewpoints and these systems is that viewpoints are objects within the system, they may be reasoned about and described just as any other description in the system. Viewpoints are not restricted to being organized into a tree structure as are the contexts of Conniver and QA4.

A key property of viewpoints is that information is only added to them and is never changed. Consider, for example, a description of an invoice. The description is in a viewpoint and may be further described in the viewpoint increasing its specificity. There may be rules that maintain constraints between attributes of descriptions, thus as information is added to a viewpoint further information may be deduced. For example, a rule for invoice descriptions may state that the subtotal plus a sales tax must equal the total; thus when any two of the attributes is known the third may be calculated. Should a description in an attribute be changed in a particular viewpoint, for example the subtotal change from $5 to $10, then the following scenario might occur:

1. A new viewpoint is created and described as being a successor to the old viewpoint.
2. All descriptions that were not derived from the changed description are inherited to the new viewpoint.
3. The new description is added in the new viewpoint, any deductions resulting from this new information are made.
4. The descriptions in the new viewpoint describe the changed state of the invoice.

In this case the new viewpoint inherits all but the changed description and the descriptions deduced from the changed description from the old viewpoint. What actions are taken when information in a viewpoint is changed is controlled via sprites. Sprites are procedures that fire when a condition they are watching for arises in the knowledge base [2]. Sprites typically fire when assertions are made or goals are posted. In the example above a simple action is specified: all information not derived from the changed information is inherited into the new viewpoint. Other actions would be to disallow change, in the case of protected information, or to signal a contradiction and allow the user to help resolve it.

6. An Example

An example is presented in the paper that illustrates the use of viewpoints in the support of organizational work. The example emphasizes the following points:

- **Problem Solving Support** - Use of the problem solving support paradigm in helping office workers in their tasks.

- **Goals** - The use of goals to describe office work. How these goal descriptions can help office workers in the performance of their tasks.

- **Contradiction Handling** - examples of Omega's contradiction handling capabilities in dealing with real work knowledge.

- **Sponsors** - used to control the amount of effort expended on a particular reasoning tasks and to reason about the reasoning task it self.

7. References

1. Fikes, R. E. and Henderson, D. A. On Supporting the Use of Procedures in Office Work. Proceedings of the First Annual AAAI Conference, American Association for Artificial Intelligence, August, 1980.

2. Kornfeld, W. Concepts in Parallel Problem Solving. Ph.D. Th., Massachusetts Institute of Technology, 1982.

3. McCarthy, J. and Hayes, P. J. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, Edinburgh University Press, 1969, pp. 463-502.

4. Rulifson, J., Derksen, J. and Waldinger, R. QA4: A Procedural Calculus for Intuitive Reasoning. Artificial Intelligence Center Technical Note 73, Stanford Research Institute, November, 1972.

5. Sussman, G. J. From PLANNER to CONNIVER - a Genetic Approach. Proceedings of the AFIPS Fall Joint Computer Conference, AFIPS, 1972.

6. Winograd T. Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. MAC TR 83, MIT, 1971.

TOWARD A NEW FRAMEWORK FOR OFFICE SUPPORT

Raymond R. Panko
University of Hawaii

Ralph H. Sprague, Jr.
University of Hawaii

ABSTRACT

Office automation is the outgrowth of one form of corporate information technology -- the office products category. If we are really to enhance the total office function, however, we will need a much broader concept of office information systems. This paper proposes one such concept, which we call office support.

THE OFFICE OF THE PAST

In the early 1960s, there was almost no technology in the office. Only the telephone was widespread, and even it did not reach every desk. In addition, toll rates were very high, so long distance calling was a rarity.

Computers were just beginning to spread in data processing applications, replacing the older electronic accounting machines. Even where computers were in use, their impact was modest. A McKinsey study made in the early 1960s (Ref. 1) found that only nine of the twenty-seven installations it analyzed were even covering operating costs, much less providing an adequate return on investment.

Xerographic copiers existed but were only beginnning to catch on. Electric typewriters were commonplace, but the first word processor would not come until 1964. Facsimile was used only in specialized applications and would not begin to flower in general office use until the 1970's.

The future for office technology, however, was extremely bright. Nearly all of the current office technology staples had been invented, and costs were beginning their steady long term fall.

This change could not come too quickly to satisfy user needs. As shown in Table 1, white collar work grew rapidly after 1960. Porat's work on the information sector of the economy (see Figure 1), moreover, showed that the total information sector spurted from 1950 to 1980. During that same period, the industrial segment plummeted. These statistics speak of a truly profound change in organizations that has been the subject of too little study.

THE CLASSIC INFRASTRUCTURE

As spending on information technology began to grow in the 1960s, ways to manage spending became necessary. Since the various technologies were almost completely independent in their operations, a fragmented approach to information management generally evolved.

In this fragmented approach, four major specializations emerged. As shown in Figure 2, each had its own products, authority center, and vendors. Most offices that used information technology heavily, furthermore, were served primarily by one of the four authority structures.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0082 $00.75

TABLE 1
THE WHITE COLLAR EXPLOSION

Category	Percent of Workforce 1900	1940	1980	1990*
White Collar	17.6%	31.1%	51.2%	51.2%
Managers	5.8	7.2	10.9	10.7
Professional/technical	4.3	7.4	15.8	14.8
Clerical	3.0	9.6	18.3	19.0
Sales	4.5	6.7	6.2	6.7
Other	82.4%	68.9%	48.8%	48.8%
Total	100.0%	100.0%	100.0%	100.0%

Sources: Refs. 2-4
*Projected

Control resided primarily in the hands of users. Computing, for instance, meant data processing (i.e., records processing) almost exclusively; so computers usually were put under the chief financial officer, as electronic accounting machines had been before them.

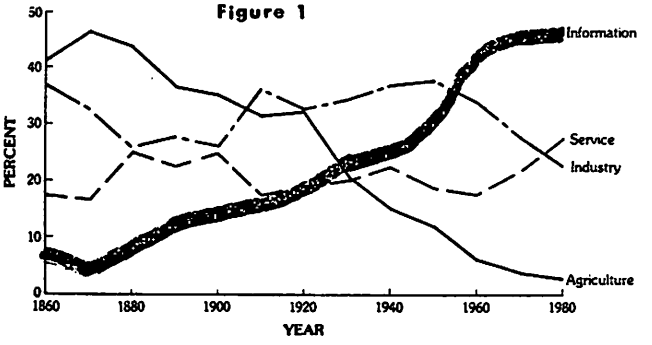
In time, of course, DP became largely independent of the controller. First, the new technology required so many specialists that a rather independent DP center emerged naturally. Later, the corporate computer clientele extended beyond accounting, payroll, and billing, further strengthening the DP center's independence.

Specialized office products for mailrooms, reproduction centers, records management centers, typing pools, and the like were generally purchased by the individual offices using them, although the administrative VP or someone like this usually had at least nominal oversight responsibilities. Although spending on these products soon became quite large, their use in out-of-sight operations gave them far less visibility than their dollar spending would justify.

Telecommunications also received much less attention than its dollar spending should have justified. The reason for this is that both telephone and Telex service were traditionally supplied by full-service vendors, who relieved user organizations of most administrative burdens. This prevented large internal staffs from growing, except where internal teletypewriter networks were implemented. By a curious bureaucratic myopia, most organizations have wrongly equated the importance of various information tools with staff effort instead of with spending or impact.

General office products were usually controlled by the administrative VP. Consisting of small ticket items such as typewriters, answering machines, facsimile terminals, and convenience copiers, general office products were usually controlled broadly, via the annual budgeting mechanism. This left considerable initiative to individual departments.

Figure 1



Percentage aggregation of the U.S. work force

When this classic four-fold infrastructure emerged in the 1960s, it made considerable sense. It placed control fairly near to users. Furthermore, since the technologies were so diverse, there was no compelling reason to create a unified information management structure.

PARADIGM LOST

While the scattered, schizophrenic infrastructure that emerged in the 1960s and 1970s once served a useful purpose, there are now strong pressures toward integrated information management.

First, spending has grown so large that scattered management no longer satisfies basic corporate accountability principles. Table 2 shows that total spending for office systems of all types was about $3,000 per office worker in 1980. It is now over $4000. Nonoffice applications, including process control and, more recently, robotics, are also important in many organizations.

FIGURE 2

THE "CLASSIC" INFRASTURCTURE

Product/Service	Typical Corporate Authority	Vendors	Typical Users (Markets)
Data Processing, Management Reporting	Director of Data Processing	IBM (DPD) DEC CDC Honeywell Burroughs	Accounting Payroll Reservations Check Processing Some Managers Some Professionals
Specialized Office Products mailing equipment duplicators mircrofilm centralized word processors	Administrative VP	Frieden Bell & Howell Kodak 3M	Mailrooms Reprographic Record Management WP Center
General Office Products copiers convenience word processors typewriters	Administrative VP	Xerox Savin IBM (OPD)	Various offices
Telecommunications telex telephones PABX	Telecommunications Administrator	AT&T GT&E	Telex Room General Offices Switchboard

TABLE 2

ESTIMATED U.S. SPENDING ON OFFICE SYSTEMS: 1980

Category		Spending ($ Billions)
Telephone		$ 36
Office Systems Support Labor[a]		$ 30
Office Data Processing[b]		$ 27
Text Processing		$ 26.8
Reproduction	$ 8.3	
Postal delivery	6	
Paper, not otherwise classified	5	
Word processing	1.5	
Typewriters	1.3	
Electronic mail	1	
Micrographics[c]	0.9	
Calculators	0.6	
Mailroom equipment	0.6	
Pens and mechanical pencils	0.6	
Miscellaneous[d]	1	
Office furniture		2
TOTAL (ROUNDED)		$ 120
PER OFFICE WORKER (ROUNDED)[e]		$3000

Notes: a. Includes the labor needed to plan, operate, and maintain office systems. Data processing center labor would fall into this category. Does *not* include end user labor.

b. Includes all data processing done to handle what would normally be considered office functions.

c. An unknown portion of this micrographics spending really belongs under office data processing.

d. Includes answering systems, dictation machines and many other items.

e. Assumes 38 million office workers.

Source: Panko (Ref. 6)

Second, technological barriers among various tools are rapidly dissolving. Most word processors and other intelligent office products are really general purpose computers that can handle many kinds of software. In October 1981, for example, Office Products News could count more than 140 products combining WP and DP (Ref. 7).

Furthermore, a number of office technologies that had previously contained no electronic logic at all are beginning to be computerized. Many PBXs can handle data communications and some can even do protocol conversion, that is, allow unlike, machines to work with one another. Intelligent copiers can serve not only as copiers but also as DP and WP printers and even facsimile terminals. Soon it may be difficult to find any significant piece of office equipment that does not have a microprocessor and cannot be programmed to handle multiple tasks cutting across old authority boundaries in the firm.

Networking is rapidly dissolving the remaining barriers around traditional bailiwicks. On the one hand, office products are increasingly being imbedded in "integrated office systems" based on decentralized networking. On the other hand, data processing is moving rapidly to distributed data processing. (In real dollars, mainframe sales are virtually stagnent.) So far, these two network-based approaches have shown few signs of merging, each being preoccupied with problems that are of only moderate concerns to the other. But in time the two approaches are almost certain to merge.

Clearly, it is time to create a new integrated approach to corporate information services. Unfortunately, during the twenty year period in which information processing was separated, the different pieces developed very strong traditions and insights that helped each serve its clientele. This raises serious problems for integration, because each tends to see integration as the extension of it traditions and insights to the other "unenlightened" parts of the information processing infrastructure.

In strategic planning, it is often said that you are killed by your past successes, not your past failures. Failures tend to be self liquidating, and they do not tend to be repeated. Things that a company is good at, on the other hand, become deeply imbedded in the firm's consciousness and become extremely difficult to dislodge, even long after their usefulness is over and they are actually damaging the firm. In the information infrastructure, data processing and office automation are proposing philosophies of corporate information management that are really extensions to their past successful patterns.

THE DP SOLUTION: INFORMATION RESOURCE MANAGEMENT

Office automation people tend to characterize DP as "numbers instead of words". The whole idea behind the coming of the phrase "word processing" was to reinforce this view of DP. But data processing is really the processing of records, which generally are a mixture of text and numbers.

Even the earliest civilizations required extensive records processing. Each of these transactions had to be added to appropriate files or repositories. All of this work was done by hand, as it would be until 1890, when Hollerith's famous punch card machine was created for the Census of 1890. In the 1930s, electronic accounting machines (EAMs), which grew up in response to New Deal programs like Social Security. Finally, as computers grew, they speeded the process of transaction processing and posting. In addition, files became complex, using hierarchical, networking, and other structures far too complex to use with earlier systems. This allowed rapid and selective information retrieval, leading to online transaction processing. It also allowed better information summarization, leading to MIS and eventually decision support systems.

But complex file structures also created problems for the DP center. Now it was more difficult to ensure that data were accurate, timely, and secure. This data management problem has become even more important today now that data base management systems have made basic file handling easier. It has even been suggested that the DP center will even-

tually do only data management, leaving retrieval and other actions to user departments. End user programming through report generation programs and IBM's Information Center idea are largely steps in this direction.

Now DP is touting an even broader idea, called information resource management, or IRM. The core idea of IRM is that information is a primary resource of the firm, as basic as cash, capital, labor, and equipment.

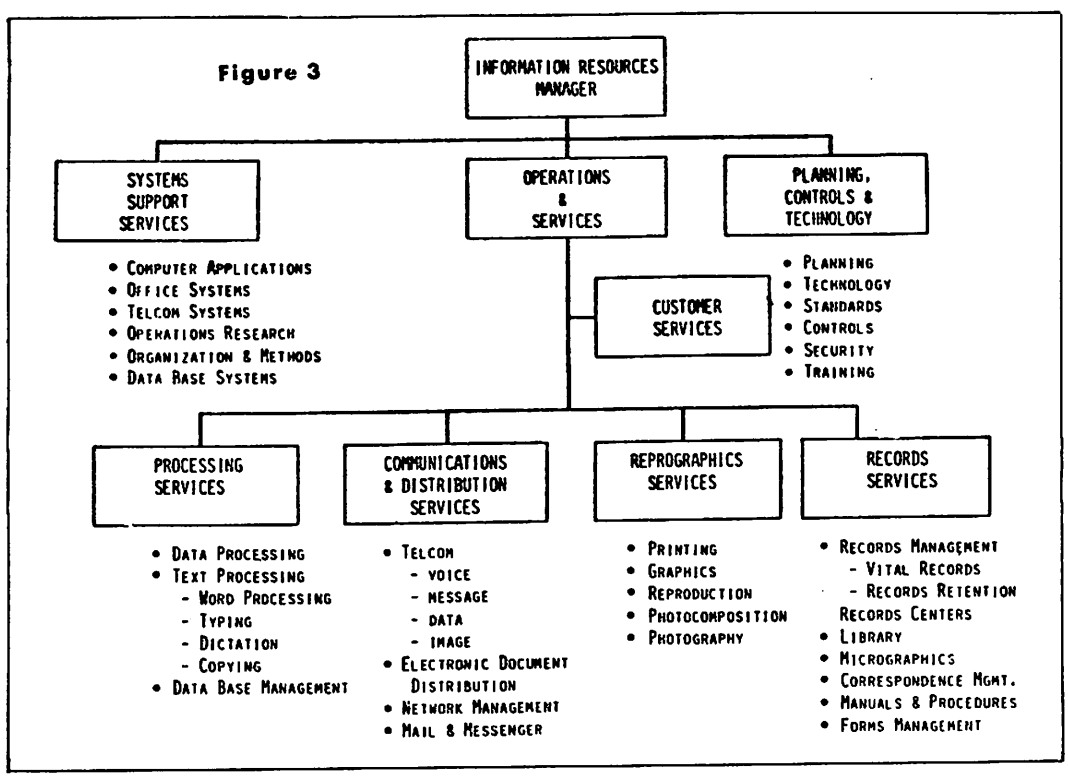
Thus, information must be managed across the firm in an integrated manner. IRM would encompass not only computer data bases but also other information repositories such as libraries.

The biggest blind spot in IRM today is that it grew out of DP's file management and data management tradition, so it tends to focus on information repositories. The IRM concept, particularly as embodied in the Paperwork Reduction Act of 1980, is rich in repository management ideas but generally less eloquent on other aspects of corporate information processing.

Proponents of IRM are likely to argue that our characterization of information resource management is incorrect, and that IRM is concerned with office automation, voice communications and other non-repository tools. To support their criticism, they can point to several excellent books and papers including OA and other tools. Applied IRM also goes beyond repository thinking, they would say. When Mobil developed an IRM plan, as shown in Figure 3, nearly all information processing functions were included.

When "broad" discussions of IRM are examined closely, however, it usually becomes clear that only repository-focused tools were considered in great depth. Indeed, discussions of office automation are usually naive. In the Mobil plan, for instance, word processing is lumped with data processing and separated organizationally from electronic mail. This odd division only makes senses if the processing vs. communications distinction were set up by a DP group, for which the processing vs. communications distinction is a meaningful and important one. Overall, although IRM claims to be a total philosophy of corporate information management, it seems to be, in practice, a DP manager's idea of total corporate information management.

Figure 3 illustrates another problem with IRM, namely its neglect of what might be called information marketing. There is no strong link to users, whose needs often cut across the neat little boxes shown in the figure. It is as if all manufacturers in the United States had banned together in a cartel, only to forget about wholesaling and retailing. This is hardly acceptable in a scheme that pretends to being a total approach to information management.



THE OFFICE AUTOMATION SOLUTION: SMALL PRODUCTS AND INTEGRATED OFFICE SYSTEMS

During the 1960s, when information technology began to spread, general office products were the poor cousins in the early information revolution. Limited to such mundane products as typewriters, mechanical calculators, and copiers, general office products lacked the pizazz of computerized EDP systems. During the early 1970s, however, office products began to get a new image, as new products appeared and older products went through dramatic technological changes. The symbol of this metamorphosis was the word processor. Although developed in the 1960s, it was after display units were introduced by Lexitron in 1972 that word processing began to grow rapidly.

Another factor was the growth of facsimile. The gradual elimination of rules restricting the attachment of devices to the telephone system, after the 1968 Carterphone decision, greatly improved the attractiveness of facsimile. Xerox responded by producing the first modern solid state facsimile devices and published its connection standards to foster the industry's growth. These efforts paid off, and facsimile grew dramatically, eventually surpassing older forms of electronic mail.

Another key development was the invasion of its copiers market by a host of companies. Of particular importance was the introduction of a new class of copiers at the very low end. Today, several copiers sell for under $3,000.

The word processor and other new developments caused office products vendors and analysts to envision an electronic "office of the future" that would be filled with exciting gadgetry. Later, new products began to sell in considerable amounts, and the term "office of the future" was phased out, in favor of "office automation".

What was really happening was something akin to what happened during the late Industrial Revolution, when fractional horsepower motors were introduced. Just as fractional horsepower motors allowed work to be freed from the centralized steam engine power plants, so that power could be brought to small shops, the new office products greatly expanded the role of small information tools in many office areas previously untouched by information technology. Before, information tools had tended to be concentrated in DP operations (accounting, check processing, etc.), reproduction centers, records management centers, and other large scale operations. A golden era of OA began, characterized by the idea that "small is beautiful".

In time, the idea of office automation was broadened to include a number of tools, such as teleconferencing and telephone services, that had not originated in the general office products world. In general, this expansion of focus was healthy, because it tended to bring a broader view of office work to the OA community, which had always exhibited a disturbing tendency to equate its tools with the totality of tools used by office workers. This equation, of course, was absurd. In fact the very term "office automation" had appeared around 1948 as a synonym for what we now call DP. Indeed, most of the books written about office automation in the 1950s and 1960s barely even mentioned general office products, which they represented such a small portion of spending on information technology for office workers.

Some OA analysts realized that DP and other information tools were used primarily by people who were undeniably office workers. They tried to explain away the traditional limited vision of OA by suggesting that OA is designed to serve "general offices". But the idea of a general office has always been a vague one, and most attempts to give examples of general offices have pointed to offices in which MIS, decision support systems and a variety of professional computational tools are also used. While it is true that OA products are used in offices generally, it remains to be seen whether a general office really exists devoid of other technology.

While it is generally true that OA is moving beyond traditional general office products, no two analysts seem to agree on exactly what it does include. Some include personal computers, others do not. Most include some forms of voice communication services, but many include only some forms. Many include small business systems, which are really nothing more than DP systems for small offices, but nearly everyone excludes large scale DP. Finally, there is considerable disagreement over whether to include specialized office products, such as large scale copiers and duplicators, mailroom systems, or large records management centers.

While there is widespread disagreement over what office automation encompasses, there is widespread agreement over the future of office automation. That future, in a word, is integration. Nearly all analysts envision the rise of integrated office systems that will link all products via local and long distance networks. Most individual workers, in addition, will have personal work stations embodying a wide spectrum of tools and providing a gateway to distant resources.

Unfortunately, the basic confusion over what tools to include under OA has carried over to the integrated office system (IOS) approach. Most IOS tools seem surprisingly like generalizations of traditional general office products.

Table 3

MANAGERS' AND PROFESSIONALS' USE OF TIME

Source	No. of Respondents	Percentage of Working Day: Face-to-face Meetings	Telephone	FTF Plus Telephone	Reading, Writing	Total Communication
Managers and Professional Combined						
Xerox	17,000	18%	5%	23%	47%	70%
Booz-Allen & Hamilton	a	--	--	46	21	67
Managers Only						
Engel, et. al.	199	27%	13%	40%	34%	74%
Stewart	160	54	6	60	28	88
Hinrichs	136	45	8	53	25	78
Burns	76	--	--	52	24	76
Horne & Lupton	66	54	9	63	24	87
Dubin & Spray	8	55	6	61	5	66
Croston & Goulding	6	56	7	63	18	81
Mintzberg	5	64	6	70	20	90
Palmer & Beishon	1	54	6	60	15	75
Notting	1	--	--	59	17	76
Professionals Only						
Case Institute	1500	--	--	--	--	53%
Engel, et. al.	130	10%	11%	21%	30%	51
Hinrichs	96	25	5	30	27	57
Klemmer & Snyder	b	35	7	42	26	68

Notes: a. 90,000 time samples
b. 3,132 time samples

In general, OA has focused on rather universal tools used by most offices. More specifically, OA has focused on communication tools. As shown in Tables 3, 4, and 5, communication dominates the work days of managers, professionals, secretaries, and clerical workers.

But communication is still only part of the day. Noncommunication activities are significant for all groups and are quite substantial for professionals, lower level managers, and staff managers. Many noncommunication activities are office-specific and even person-specific, for example financial analysis in planning groups. In general, OA has been slow to understand the noncommunication and specific activities of office workers, relegating these activities to other bailiwicks.

Real productivity and performance breakthroughs, however, are likely to stem from specific tools rather than from general communication tools. At most large banks, for example, financial management systems have revolutionized asset and liability planning, and many other office functions in bankng and other industries have undergone equally critical work revolutions thanks to information technology.

Although OA has suffered from conceptual blindness, it has brought critical insights to corporarte information management. First and most importantly, it has focused on communication, thus counterbalancing the DP/IRM tendency to focus on information repositories.

Second, OA has emphasized the importance of small scale tools for individual offices and even individual workers. This too serves as a counterbalance, moderating DP/IRM's tendency to focus on large scale corporate repository policies. Even when OA has focused on repositories, it generally has focused on smaller scale repositories.

ANOTHER SOLUTION: OFFICE SUPPORT

Our belief is that a true theory of integrated information management must transcend the insights of the individual

TABLE 4
THE SECRETARY'S DAY

Face to Face Communication		9.8%
Conferring with principals	4.3%	
Taking shorthand	5.5	
Telephone		10.5%
Reading and Writing		5.2%
Reading	1.7%	
Writing	3.5	
Typing		37.0%
Other Paperwork		34.49%
Mail handling	8.1%	
Copying or duplication	6.2	
Filing	4.6	
Proofreading	3.9	
Pulling files	2.8	
Collating and sorting	2.6	
Keeping calendars	2.6	
Pick up and delivery	2.2	
Bulk envelope stuffing	1.4	
Other		3.4%
Using equipment	1.4	
Other	2.0	
Total		100.3%

N=115
Source: Engel, et. al.

technology/application compartments of the past. The best way, we believe, is simply to define what offices are, then ask what they need (regardless of past technical boundaries), and finally develop an infrastructure to support these needs. We call this approach, simply, office support. (A full theory of integrated information management would also include industrial process control, robotics, and other nonoffice functions, but we will not consider these in this paper.)

Essentially, we are redefining things in terms of markets, instead of products. When industries are young and growing rapidly, most taxonomies are based on product distinctions. Later, as growth slows, product distinctions blur as vendors enter one anothers' product categories (IBM getting into telecommunications, etc.). Furthermore, customer acumen grows, and market segmentation becomes very important. The solution in most industries is to develop market-based strategies tied to market segmentation tactics. We seem to need such a transition today in corporate information management.

As a market, the office is actually too big. With 40% of the U.S. work force spending their days in offices, it is almost certain that the office "market" is really an amalgamation of market segments, each with different needs and wants.

Pending proper marketing research to measure office segmentation, we have intuitively indentified two basic types of offices whose needs are likely to be quite different. We call the first the Type I office. It handles the basic routine information processing functions of the firm. Examples would be accounting, payroll, and billing offices, check processing operations, word processing centers, reprographics centers, records management centers, and reservations centers.

Type I offices are characterized by high volumes of transactions. The focus of management is productivity -- doing more with fewer workers. Administrative management specialists have traditionally focused on Type I offices, calling them administrative offices because they handle general adminstrative functions. When office management is spoken of in text books, Type I offices are usually being discussed. Such techniques as cost of information analysis (Ref. 23) and work design have made the management of Type I offices similar to manufacturing production management. In fact, the clerical worker in a Type I office is very much like a laborer in a paperwork assembly line.

Although Type I offices are staffed largely by clerical workers, their managerial supervisors comprise a good percentage of the firm's managerial corps, and professionals are often found in these offices as well (accountants, draftsmen, etc.).

TABLE 5

THE CLERICAL WORKER'S DAY

Category		Percent of time
Face to face		1.9%
Telephone		9.2%
Paperwork		78.5%
Creation	41.9%	
Filing	16.1	
Calculating	10.3	
Using a computer terminal	6.3	
Copying, duplicating	3.9	
Other		10.4%

N = 115

Source: Engel, et. al., (Ref 11)

We call the second type of office in our intuitive taxonomy the Type II office. It is staffed primarily by "principals", that is, managers, professionals, and nonclerical sales workers. These offices set policy for the firm and handle professional functions such as law, engineerng, finance, and corporate planning. Of course these Type II offices usually have significant clerical staffs.

Type II offices are normally judged on the basis of performance, that is, success in selecting and meeting goals, not just in terms of productivity. At the same time, there is a growing realization that these offices must become more productive (efficient) if managerial and professional salaries are to grow in the future (Ref. 24).

Note that the split is not based on individual jobs, such as manager, professional, and clerical workers. Managers in Type I offices generally have very different kinds of jobs than those in Type II offices. Clerical work also varies remarkably, being tightly controlled in Type I offices and freer in Type II offices. Even equipment used in both types of offices may differ considerably depending upon what kind of office will use them. A word processor in a production-oriented Type I word processing center must face very different demands than a convenience word processor in a Type II office.

By and large, we have the tools we need to analyze the needs of Type I offices and to manage them. Systems analysis, when married to industrial engineering tools is an efficient way to define needs. The process-oriented language used in these fields seems natural to both analysts and end users. The managerial tools in the administrative management literature, in addition, are voluminous and tested; in addition, when computers are used in the automation process, many long standing administrative management techniques that had previously been too expensive because of the measurements they required suddenly become cost effective.

For Type II offices, needs assessment and information management are in a less developed state. The language of processes and procedures is difficult to apply and seems foreign to users, who tend to think instead in terms of threats, opportunities, goals, strategies, tactics, and political maneuvering. Even if low-level processes could be identified and automated, moreover, it is not clear that performance would be enhanced as a result. The whole thrust of current efforts to develop office modeling languages may be much less attractive than their proponents claim.

Another problem in serving Type II offices is that needs are comparatively small in scale and often have limited duration. Classical systems analysis tools are far too expensive to use for small scale operations. Rather than just scaling them down, we probably should be thinking of radically new ways to understand user needs and develop appropriate solutions for small scale situations.

While Type I offices have been written about and studied in depth, Type II offices have remained "the familiar unknown". We all work in them we talk about them every day, and yet our empirical knowledge about them is small. In 1965, Horne and Lupton (Ref. 16) complained that texts on management tended to be filled with principles instead of research findings based on empirical study. Today, a generation later, this general critique remains sadly true, despite some empirical work during this period. Our systems continue to be designed to stereotype instead of designed to need.

In all likelihood, the Type II office type is too general an abstraction to have much value. Needs are probably quite specific to individual offices, so the principal office segment is probably a collection of many smaller segments. Research is badly needed to shed light

ORGNIZING FOR OFFICE SUPPORT

The Marketing System

IRM and other philosophies of corporate information management argue for the unification of all corporate information suppliers, as exemplified in Mobil's IRM chart shown as Figure 3. But as noted earlier, Supply is only one of three factors needed to make a market. The other two are users and a "marketing system" to link suppliers with users.

One function of any marketing system is to promote the products being supplied, and this must certainly be the case in corporate information management. But promotion is only part of what a marketing system does. In the commerical world, the marketing system also acts on behalf of end users, forcing changes on suppliers. This proactive role is most obvious when wholesalers and retailers are independent of manufacturers. Independent middlemen execute very effective control by selecting what products they will or will not carry. Unusually powerful middlemen, such as Sears, even have products built to their specifications.

Until now, we have given comparatively little thought to filling the ground between supplier groups and users in the corporate information environment. This is understandable, because, in the past, each segment in the corporate information infrastructure had its own markets and a fairly limited range of products. In DP, for example, systems analysts served the middleman role, while in

office products there was sufficient excess demand that promotion and tailoring were negligible concerns compared to spending control. Now, however, promotion is needed to speed the adoption of many products. Tailoring too is needed, in everything from inexpensive personal computers to complex electronic funds transfer systems.

Besides "matching" information users and providers in the corporation, the marketing system has certain broader responsibilities. First, it is likely that a department's needs will require the integation of products from various supplier groups within the firm, into unified service packages. For example, a program using both OP and word processing might be needed. The marketing system either must provide this integration (this would require the presence of significant technical expertise in the marketing system), or must ensure integration in some other way.

Also, as Robert Vierck of the Dillingham Corporate has pointed out (Ref. 25), the marketing system may have to audit existing systems, to make sure that technology is well matched to problems. DP people, for instance, might tend to have solved problems on mainframe computer when personal computers or minicomputers might have been better delivery systems.

Coping with Small Problems

However the corporation decides to build links between information suppliers and users, it will have to face up to the problems raised by the growing diversity in the sizes of computer problems appearing in corporations.

Systems analysis, for instance, was designed for large problems. As current used, furthermore, it seems to be adapted primarily to large procedurally-rich problems such as transaction processing systems.

At the department and individual levels, however, problems tend to be smaller in scale, and expensive design approaches cannot be used cost-effectively. Just as small computers are now being sold through retail stores, companies are now establishing Information Centers, a term coined by IBM, to provide "storefront" service to small users.

IBM's Information Center concept, however, evolved out of the MIS tradition and has been used most often for problems that can be solved using report generators and other tools the users can use themselves. In medical terms, the IC concept espoused by IBM is like a self-help medical clinic, in which consultants advise but do not treat.

In the medical world, we have a spectrum of service providers, including specialists, family practitioners (who can treat most ills), paramedics (who treat minor problems), and the self help clinics that merely advise. The goal is to match a problem's size with the level of resources needed to solve it. In corporate information management, we have created only the extremes so far. The traditional systems analysis approach is a high powered set of tools for large and complex tasks. The Information Center is like the self help clinic. We must now build things in between.

The Management Issue

At the beginning of World War II, Germany's U-boats were relatively ineffective at stopping British shipping. Then they adopted "wolf pack" tactics, attacking in coordinated groups. Very quickly, without any change in technology, the U-boats became a deadly force. The Allied counter offensive at sea, similarly, was comparatively ineffective until destroyers began to work in "hunter-killer teams".

These examples illustrate a simple but profound truth, that many and perhaps most "technological" revolutions are really "organizational engineering" revolutions, in which new ways of organizing work are the keys to success. Feudalism, the factory system, and mass production did not evolve; they were deliberately invented by a relatively few individuals. Charles Martel, for example, built feudalism in France in the space of a very few years.

DP and OA analysts have tended to shy away from organizational engineering after bad experiences with the original meaning of "word processing", which was invented in 1965 by IBM's German division to denote a radical reorganization of the secretarial function. In addition, few technologists have any training in the management of organizational functions, and this probably has strenthened the information professional's aversion to "meddling" in organizational matters.

If an organization does decide that its problem is how to improve the productivity and performance of office functions, not just supply technical systems, however, if must find a way to hire competent organizational analysts or refrain its own people (it cannot merely say three times "thou art an organizational analyst" to a systems analyst and expect to see results).

Most four year universities, unfortunately, have now removed "administrative management" courses, which deal primarily with the effective management of Type I offices, from their curricula. These courses are now taught primarily at junior colleges, at a very low level. For Type II offices, even less of a body of a training infrastructure

It should be possible, however, to create a course for the management of Type I operations from the existing literature. Administrative management, despite its current ostracism, has a rich body of literature from which to draw. There are also many recent strategic efforts to draw upon, including work at Xerox in cost of information analysis (Ref. 23) and at American Express in impact-driven goal selection (Ref. 26).

For Type II offices, the problem is much harder. The Critical Success Factors approach developed and popularized by Rockart (Ref. 27), Peter Drucker's work on external contribution analysis (Ref. 240 and other promising gems are scattered through business and technical writings, but they have not yet been brought together.

Overall, although the issue of how to rethink organizational work to take advantage of new conditions and new technology must lie at the core of any serious office support effort, this is the area about which we know the least and in which research is needed most urgently.

CONCLUSION

In the past, responsibility for information technology was widely scattered throughout user organizations, and no one had the ultimate authority for making office functions more productive or better performing.

Today, IRM and office automation both offer "integrated" solutions, although in practice, both approaches tend to be flavored heavily by the thinking of individual segments of the corporate information infrastructure.

What we need now is an approach to total office support, which is marked by three ideas. First, there is a "market orientation"; this means that the focus in analysis is on the work being done in the department, and support provided using whatever tools are appropriate. Second, an organizational infrastructure is established to link provider groups to end user groups in a systematic, effective, and low cost way. Third, the OS staff does not merely automate or mechanize existing processes; it also helps user departments reorganize to work more effectively, and it may even reorganize existing boundaries between functional units.

REFERENCES

1. John T. Garrity and V. Lee Barnes, "The Payout in Computers: What Management Has Learned About Planning and Control," Chapter 4 in William F. Boore and Jerry R. Murphy (eds.), The Computer Sample Management Perspectives on the Computer. McGraw-Hill Book company, New York, 1968.

2. Bureau of the Census, Historical Statistics of the United States, Colonial Times to 1970.

3. U. S. Department of Labor, Bureau of Labor Statistics, Employment and Earnings, Vol. 27, No. 2, February 1980.

4. U. S. Department of Labor, Bureau of Labor Statistics, "Occupational Projections and Training Data, 1980 Edition," Bulletin 2052, September 1980.

5. Marc U. Porat, The Information Economy, U. S. Department of Commerce, Office of Telecommunications Policy, 1977.

6. Raymond R. Panko, "Spending on Office Systems: A Provisional Estimate," accepted for publication in Office Technology and People. Some data from this paper were published in Raymond R. Panko, "A Different Perspective on Office Systems," Administrative Management, pp. 30-32, 75, August 1981.

7. Anon , "Integrated WP/DP Systems," Office Products News, October, 1981. 8. Marcia Blumenthal, "Mobil Considering IRM as Separate Line Entity," Computerworld, p. 27, February 9, 1981.

9. Xerox Corporation study cited in John Ganz and James Peacock, "Office Automation and Business Communications," Fortune, pp. 7-80, October 5, 1981.

10. Wayne L. Rhodes, Jr., "How to Boost Your Office Productivity," Infosystems, pp. 38 ff. pasim, August 1980.

11. G. H. Engel, J. Groppusa, R. A. Lowenstein, and W. G. Traub, "An Office Communications System," IBM Systems Journal, Vol. 18, No. 3, pp. 402-431, 1979.

12. Rosemary Stewart, "Managers and Their Jobs," (Pan Books, London, 1967).

13. John R. Hinrichs, "Communications Activity of Industrial Research Personnel," Personnel Psychology, Vol. 17, pp. 194-204. (November 1963).

14. Tomas Burns, "Management in Action," Operational Research Quarterly, Vol. 8, No. 2, pp. 45-60 (1957).

15. James Horne and Tomas Lupton, "The Work Activities of Middle Managers -- An Exploratory Study," Journal of Management Studies, Vol. 1, No. 2, pp. 14-33 (1965).

16. R. Dubin and S. L. Spray, "Executive Behavior and Interaction," Industrial Relations, No. 3 (1964).

17. J. D. Croston and H. B. Goulding, "The Effectiveness of Communication at Meetings: A Case Study," Operational Research Quarterly, Vol. 17, No. 1, pp. 45-47 (1967).

18. Henry Mintzberg, "Managerial Work: Analysis from Observation," Management Science, Vol. 18, No. 2, pp. B-97-B-110 (October 1971).

19. A. W. Palmer and R. J. Beishon, "How the Day Goes," Personnel Management, 1970.

20. Notting, reported in Sune Carlson, Executive Behaviour (Stromberg, Stockholm, 1951).

21. "An Operations Research Study of the Scientific Activity of Chemists," Case Institute of Technology (1958).

22. Klemmer and Snyder, cited in J. E. Karlin (ed.), "Human Factors Research -- Some Recent Findings and Future Problems," Proceedings of the 5th International Symposium on Human Factors in Telecommunications (1970).

23. Paul A. Strassman, "Managing the Costs of Information," Harvard Business Review, pp. 133-142, September - October, 1976.

24. Peter F. Drucker, Managing in Turbulent Times, New York, Harper & Row, 1980.

25. Robert K. Vierck, Dillingham Corporation, personal communication, December 1981.

26. Business Week, "Boosting Productivity at American Express," pp. 62,66, October 5, 1981.

IMPLEMENTING OFFICE AUTOMATION:

PRINCIPLES AND AN ELECTRONIC MAIL EXAMPLE

Jeffrey H. Tucker

Department of Quantitative Methods
College of Business Administration
Georgia State University
University Plaza
Atlanta, Georgia 30303

ABSTRACT

This paper discusses reasons for resistance to office automation and proposes a method of implementing change drawn from the literature on organizational development and on the implementation of management information systems. The general method is applied to a pilot project of an electronic mail system. Data gathered from the 36 project participants are summarized. Although the project data indicate a need for an EMS, the project failed to produce an adequate user communication volume, and it was cancelled. The paper describes the problems which prevented the effective utilization of the system.

PRINCIPLES

In the last ten years, office productivity has increased at less than one-tenth the rate of industrial productivity. By increasing the capital outlay per office worker it would seem that automation could increase productivity in the office as it has in industry. However, installation of word processing (WP) systems, electronic mail systems (EMS), electronic conferencing or management information systems (MIS) has often failed to produce the expected gains.

One reason for the disappointing results is that while there are some similarities between industrial and office automation, there are also differences. The two kinds of automation differ, for example, in their orientation. Industrial automation is task oriented and focusses on increasing the speed of production. Office automation can speed the production of memos and reports, but its real focus is goal oriented. Its goal is effective information exchange, which involves human perception and interpretation of raw data into information suitable for decision making.

Another difference between industrial and office automation is the way in which their results are measured. Industrial automation can be measured by production rate. Office production too, can be measured by production rate, but a more important variable to measure is the effectiveness of information exchange and decision making. Quantity is easily measured, but quality of information is a much softer variable and therefore harder to measure. It is difficult to put a financial value on information that shows up on the bottom line as neatly as increased task performance.

If the change to office automation is a fact of organizational life and if the variables which monitor the change are hard to measure, then the method of negotiating that change should be carefully chosen. Two methods are often cited: top-down (authoritarian) and bottom-up (democratic). Neither is necessarily good or bad; each has its advantages. The top-down method is fast: the decision to change is made by those with the best grasp of the organizational environment and resources. The implementation will succeed to the extent that the users follow the behaviors prescribed for them. The bottom-up method utilizes the knowledge of those employees who are doing the work. They have a detailed knowledge of the task, but have to be given the over-all picture of organizational resources and environment in order to participate effectively in the change process. The democratic method takes more time but produces greater commitment to the change, therefore requiring less management control.

An individual attempts to predict and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0093 $00.75

control his environment. Any externally determined change which introduces new variables produces anxiety and resistance. In order for management to implement office automation effectively, the reasons for an individual's resistance to change must be explored and dealt with.

REASONS FOR RESISTANCE

Four areas of concern account for a large part of the resistance to office automation. Each area should be considered before implementing any system.

1) Change = anxiety

The change to an automated office threatens to disrupt the existing environment, forcing people to learn new tasks and dislocating established norms, roles, interactions and levels of status. People become anxious about disruptions in their environment. The greater the perceived change, or the fewer the facts known about it, the greater the anxiety it produces and the more likely it is that the change will be overtly or covertly resisted. Knowing that an organizational change is planned but not knowing specifically what is going to happen can lead to employee disatisfaction and an attempt to unionize (6).

2) Automation = loss of jobs

The term "office automation" is an unfortunate one, since historically automation has been associated with the replacement of workers by machines in order to increase efficiency. The popular press has used the terms "office of the future," "electronic office," "paperless office" and "office automation" interchangeably, and the average office worker treats them with equal distrust. Whisler's study (23) shows that the installation of large computers in the insurance industry eliminated 60% of its clerical staff. Failing to take into account the possible loss of jobs will produce massive employee anxiety. The impact on jobs that would result from office automation should be dealt with before the organization is committed to a new system. If the employees see that the system will be implemented before the questions of transfer, training, or "out-placement" have been addressed, they may be more concerned with protecting their jobs than with implementing a system which threatens their security. It is obvious that clerical personal can fear job loss through office automation but perhaps less obvious that managers who function as information channels also can have their jobs threatened by an electronic information system.

3) Automation = loss of freedom

Interaction with a machine which has specific input and output formats can be seen as restricting behavior. For example, depending on the system and the method of implementation, WP can be seen as creating documents more effectively or as reducing secretaries to keypunch operators. Some people dislike replacing human interactions with electronic messages, which omit such non-verbal information as voice tone, facial expression and posture. Increased reliance on electronic information channels also lessens individual face-to-face and voice-to-voice contact, thereby limiting the range of social contact.

4) Information = power

Information is often withheld in order to establish power and influence in organizations. An electronic information system (whether it be WP, EMS, or MIS) raises the question: Who will have access to what information for what purpose? There are great opportunities for monitoring, control and invasion of privacy in any of these systems. Information is a political resource; rules governing its use need to be clearly understood. Computer-based systems can, for example, be used positively, to relieve employees of tedious checking and to provide feedback to develop employee skills, or punitively to track employee productivity and errors for disciplinary action. Whisler (23) concludes that the large-computer technology in the insurance industry is sufficiently powerful to change an organization in predictable ways: to centralize, to shift power to the computer group, to reduce the number of clerks and middle managers, to shift control to the machine, and to decrease communication between people. Changes in information technology radically change the distribution of power in organizations including the structure of the organization itself.

Office automation is new, but the problem of organizational change has been a subject of study for over twenty years. Sharing information with potential users and involving them in the change design has been shown to lower resistance and facilitate implementation (21). Bjorn-Anderson's study (5) of automation in five European banks suggests that the clerks' reaction to automation was more often a result of the implementation and use of the technology than of the technology itself. A method of implementing office automation can be drawn from the longer history of organizational change and development.

A METHOD FOR CHANGE

Many office automation projects are implemented after an upper management decision to increase office efficiency by utilizing the latest technology. The

anticipated efficiency, however, often fails to materialize because of "people problems." To avoid these problems it is essential to assess the users' tasks, attitudes and skills and to plan the way in which the change will be managed before the system is completely designed and implemented.

The users' reaction to any change will be based upon their past experience with organizational change. Organizations often espouse a participative or democratic style of management, but when faced with problems, time pressure or a crisis, their theory in use regresses to an autocractic or dictatorial one. Participative change requires more time in the initial planning, but has been shown to produce a smoother implementation with a higher probability of realizing the hoped-for gains in effectiveness (3,6,14,15,21).

The managers of an organization should review their past style of change, successes and failures. They should consider the resources they will commit to meet workers' needs for security, equity, learning and participation. Are employees to be treated the same or will allowances be made for individual differences in needs, attitudes and skills?

To lower employee resistance to change, a plan for change should contain the following four steps:

1) Prepare a philosophy statement which specifies the reasons for change, the management's understanding of the environment surrounding the change, and management's commitment to the people affected by the change.

2) Analyze the existing workflow to determine which technical system will produce the desired change.

3) Analyze the existing social system's function to determine how it can be best integrated with the technical system.

4) Begin an iterative approach that balances the technical system (which requires operators) with the social system (which should provide opportunity for individual security, equity, learning and participation).

This plan for change is not new or unique. It is derived from the literature of organizational development (OD) and behavioral psychology (9,10,18,21). The literature on office automation is often descriptive (12,20,22) and is not as prescriptive as that on transaction processing or management information systems, which have been in use longer. The ideas of technical system analysis, social interaction and user participation, however, have been well researched, particularly in the field of MIS. Many studies have looked at the factors affecting system success and failure and have attested to the importance of the principles mentioned above. Argyris (3) sees MIS as threatening to increase upper management's control over organization members, a threat which tends to decrease the kind of information exchange most needed to define, design and implement the system. Zmund (25), focusing on individual differences in users, suggests that their preconceived attitudes predict MIS usage and their involvement with the system design predicts their satisfaction with it. Ginzberg (11) finds that an examination of pre-implementation inaccuracy in user expectations for a system can predict the system's success or failure. Alter and Ginzberg (2), in analyzing over seventy systems, isolate eight risk factors and sixteen risk reducing strategies in a seven-step system development model. Harvey (13), after analyzing thirty-one system designs, isolates characteristics of the problem, of the company management, and of MS design teams, and concludes there is a need to indentify the human and organizational problems leading to system failure or success. Zand and Sorenson (24) find Lewin's psychological theory of change (unfreeze, move, refreeze) to be a valid model for system development. Kling (16), while stating that talk about involving users in design is cheap, proposes five policies for fostering a user-centered attitude in the design group.

The implementation of office automation appears to be based on little or no theory and seems to follow the general philosophy of "bring the new system in and have the employees use it." While office automation systems are relatively new, the older OD and MIS literature stresses the importance of user participation in the planning and design stages of introducing a new system in order to decrease resistance to the change and increase commitment to the system. The explanation of reasons for resistance and the method for change outlined above utilize the insights of OD and MIS research literature. To ignore this body of literature when implementing office automation is to risk committing the sins of the past as well as to waste time reinventing the wheel.

AN EMS EXAMPLE

A large corporation was considering installing an EMS to serve over 2000 employees. The system would allow messages to be entered and received on computer terminals; to be distributed; to have notes appended; and to be filed, sorted or scanned according to author, date or

subject. To test the possibilities and problems of an EMS, a pilot study was planned with 30 to 50 users. The objective of the system was to reduce the time wasted in trying to reach someone by telephone or by a conventional office-mail system.

The reasons why people resist change and the four-step method for change presented in this paper were discussed with the staff group planning the pilot study. The following ten steps were suggested to them as an application of those principles to their pilot study:

1) Prepare a philosophy statement for the study, outlining the reasons for considering an EMS and addressing any anticipated areas of resistance.

2) Determine the potential users' needs, skills and attitudes toward an EMS with a survey questionnaire (1,3,17,19).

3) Conduct a communications audit (7,8) of the frequency of usage of existing communication channels (telephone, memos, meetings, reports); the direction of communication (supervisors, peers, subordinates); and the length of messages on each channel, using a tally sheet for one week.

4) Evaluate the existing EMS systems available against the data obtained through items 2 and 3. Discuss the alternatives with the users' supervisors and select the best alternative.

5) Meet with all the potential users to discuss the philosophy statement and the proposed training and support for the system. Solicit comments, concerns and possible problems from the users at a second meeting.

6) Prepare the final integration of the system and the users by revising the system on the basis of the users' comments, preparing the training in detail, and defining the placement of the system hardware.

7) Begin the start-up in stages. Pick the nodes in the communication network which have the greatest need and the largest number of users, so it will be more likely that at least a few people will want to use the system, will respond to the training and will be able to train the others.

8) Gather weekly usage data. Feed this data back to the users. Solicit the reporting of problems with the system and provide the required support.

9) After the EMS usage has stabilized, measure the users' attitudes toward the system with a questionnaire and conduct another communications audit to measure the changes in the pattern of communication between the available channels (telephone, meetings, written and EMS).

10) Analyze the pilot study and make recommendations to the corporation. The system should continue to be monitored for possible improvements.

The research design made use of pre- and post-project questionnaires to determine job characteristics and attitudes, together with five-day tallies of actual communications before and after experience with EMS. Judging from the currently available literature, this project would have provided a more complete understanding of EMS usage and impact than previous studies. The effect of adding an EMS communication channel to written, telephone and face-to-face channels could be measured, and it was hoped, be understood. The project was designed to aid the corporation in deciding whether an EMS would increase the effectiveness of communication for its entire staff of 2000 employees.

A BRIEF HISTORY OF THE PROJECT

The pilot study was initiated in August 1980. A proposal with a ten month schedule, which was agreed to in September, included most of the ten suggested steps. An updated version was written in February 1981. The responsibilities of the corporation and the consultants were modified in March. Two phases were planned for the consultants' participation. The first phase was to produce a questionnaire and a communication tally sheet. The second phase was to provide consulting assistance during the implementation and start-up of the system and to provide a report on the data gathered by the consultants from the questionnaire and interviews with the corporation employees.

The questionnaires and tally sheets were distributed to 36 employees at a introductory and preview meeting in September 1981. All of the 36 employees were involved in planning new products. The questionnaires and tally sheets were to be completed and returned in a week, after which the EMS system training sessions would be held. Ten days after the training sessions, fewer than half the expected questionnaires had been returned. A memo to encourage the completion of the survey data was sent at the end of September. Group meetings were held in early October to discuss the EMS usage and to feed back the initial results of the survey data. Data on system usage was available only on a monthly basis. In November the EMS trial was canceled because of low usage during October. In early December individual interviews were held with 20 participants to determine the

cause of the EMS project's failure.

DATA GATHERED

1) Pre-project data

A week after the training sessions only 17 questionnaires and 12 tallies had been received from the 36 participants. These respondents felt the timeliness of the interoffice mail was inadequate, that there were problems with "ping-pong" telephone calls and the filing/retrieval of documents. Open-ended questions at the end of the questionnaire pointed to concerns about the availability of terminals and how often the members of the network would check the system for messages.

An indication that the EMS interaction would be low was found by a comparision between the eight persons most frequently contacted by each respondent listed on the questionnaires, the actual communications listed on the tally sheets, and the 36 people listed on the EMS network.

Although the initial data indicated a need for an EMS, the low number of questionnaires returned indicated a possible low commitment to the EMS trial.

2) Group meetings

After another ten days, meetings were held with groups of EMS users to share the initial data and solicit comments and concerns about the EMS. The major problems reported with the EMS were: the lack of availability of terminals, the time required to logon the system, the composition of the network, the typing speed of the users, the software, and a move to a new building. Fewer than half the persons on the EMS network attended one of the three meetings.

3) Final questionnaire data

After another 15 days (five weeks after the survey forms were distributed) the 26 questionnaires (from 36 participants) which had been received were analyzed. Four of the questionnaires were from persons who supervise four or more subordinates; fourteen of the 26 reported no subordinates. Work satisfaction was measured in 17 areas. The respondents were most satisfied with the quality of the relationships with their co-workers and their communication with subordinates. They were least satisfied with the adequacy of document filing/retrieval, the adequacy of administrative support and the timeliness of information received from superiors. Several questions asked for descriptions of the respondents' present job situation. The respondents said that their jobs required considerable interaction with others and were often interrupted by unimportant telephone calls. They used a computer terminal less than once a month, although they had all used a terminal or typewriter in the past, at an average typing speed below 20 words per minute. They are, on the average, neutral toward using a keyboard as a part of their job. The EMS participants have high needs for communication but have problems in reaching people by telephone. Over 40% of telephone calls are not completed on the first attempt. Between two and three attempts are required to complete the call. Other questions surveyed the distribution of written documents. Over 40% of the documents are passed on without any changes. Comments on a document's content are made to about half the documents received. Over two-thirds of the documents created are for distribution within the company: half within the building and half outside the building. Distribution lists with six or more addresses are used twice a week or less freqently. The respondents felt that the EMS would probably replace some memos and telephone calls, but would be less likely to replace face-to-face meetings. Most of the respondents expressed interest in the EMS.

Although fewer than three-quarters of the people on the network turned in a questionnaire, the data received strongly support the contention that an EMS was needed and would have increased the effectiveness of communication.

4) Tally sheet data

Analysis of the communication tally sheets indicates that 47% of all telephone calls initiated and completed lasted less than three minutes, 48% lasted between three and 15 minutes, and five percent lasted longer than 15 minutes. The total number of telephone calls per week range from seven to 65. The average participant made five incomplete calls per day (failed to reach the called party). Face-to-face interactions were only half as frequent as telephone communications.

5) Post-project individual interviews

In order to understand the reasons for the low usage of the system, which caused its cancellation two months after its inception, interviews were scheduled with 20 project participants. Following is a list of eight problems associated with the project:

a) Availibility of terminals

The system was designed with individual IDs, which implies that each user has access to a terminal. However, because individual terminals were too costly, many participants had to walk to another room to borrow or use a terminal. In the first week of operation, the system

was inoperable twice, and an insufficiency of dial-in ports frustrated the potential users.

b) Inactive recipients

In order for an EMS to work, the participants must check into the system on a daily basis so that the sender of a message is assured of a timely reply. Whether from lack of commitment to the project or from the unavailability of terminals, the project never reached "critical mass." For example, one team leader called a meeting with the EMS. Instead of the expected 12 to 15 people at the meeting, only three people showed up! In order to start up a successful EMS, the number of senders, receivers and relevant messages must increase over time. The interviews indicated that most participants who used the system in the first two weeks decreased rather than increased their usage over time because of too few relevant messages on the system and the difficulty of accessing the system.

c) The move to a new building

The original EMS was to link individuals located in different buildings. After the move to a single building, many participants were sufficiently close to each other that communicating face-to-face took less time than accessing the EMS. The move also produced a reason to lose contact with the EMS -- a contact that was never reestablished. The move also changed the location and therefore the availability of terminals.

d) System software

Several participants felt that the software of this particular EMS was not up to the state-of-the-art. Since the EMS was perceived to be in competition with the telephone, having to spend more time than is required to dial ten numbers on a telephone was seen to be a detriment to the EMS. Some indication of a message waiting or the printing out of a message upon receipt would have been preferred.

e) Network

Many of the participants stated that a relatively small percentage of their volume of communication was with members of the EMS network. This fact was borne out by initial analysis of the questionnaires, the tally sheets and the network.

f) Commitment

Several participants said that the system was never "sold" to them, so they gave the EMS a relatively low priority among their required activities. There was a tendency to stop using the system rather than to work on resolving the problems with the system. Upon closer questioning it appeared that the problems of insufficient terminals and the difficulty of accessing the system were seen as impossible to solve.

f) Reference material

Two of the participants mentioned that they often attached reference documents to their written communications; since such reference documents were not filed in the system, these participants felt the system did not meet their needs.

g) Type of communication

Communications can be thought of as two types: structured and unstructured. Structured communications consist of such messages as requests for information, a meeting announcement, or a status request for a specific item. Unstructured communications are primarily exploratory and most often appear in bargaining, negotiation and problem solving situations. Most of the upper level managers felt that their communications were primarily unstructured. These managers felt that an EMS was not particularly suited for their communication needs.

CONCLUSION

A pilot EMS project in a large corporation failed. The project was not completed as designed because of problems with: terminal accessibility, a move to a central building, the system software, the composition of the network and low commitment to the project. Although almost all of the participants interviewed after the demise of the project felt that an EMS would improve the effectiveness of their communication, they believed the particular system under evaluation was not the one to meet their needs.

To analyze the failure of the project, the principles stated earlier in this paper should be reviewed. The four reasons for resistance (anxiety; job loss; loss of freedom; change in power or information flow) were not as important as they might have been in the installation of a WP or MIS system. Although any change requires some new learning and anxiety, there was little indication that the EMS was seen as a threat to job security, job freedom or as a shift in the information flow or power structure. The EMS was a relatively low threat introduction of office technology and should have produced little resistance from the users. Some time-management systems have failed because managers did not want their lack of activity exposed. Although it was not possible to check this hypothesis, it is possible some of the persons in the project did not return a questionnaire or

tally sheet because they did not want report low communication activity.

The four-step method for change (philosophy statement; workflow analysis; social analysis; interative implementation) was not actually followed. In retrospect, it appears that the management was not perceived by the users to be committed to the pilot project. Developing a philosophy statement would have clarified the objectives of the project for upper management and increased the participants' perception of the project's importance. The preliminary analysis of the technical and social systems was made by the staff group planning the change, with little participation by the users. In retrospect, it is evident that the network did not include a set of highly interacting individuals and that if the number of existing terminals were not sufficient or were not conviently placed, the acqusition of additional terminals was left up to the participating groups. The workflow and the social system that the EMS were to serve were inadequately analyzed and understood. The system was not designed or started up in an iterative manner, but selected and made available as a package. Due to constraints of time and money the pilot project was a test of a particular commercially available EMS, not an EMS designed or tailored specifically to the users' needs. It would appear the project made availible a system which provided a new technology to accomplish existing tasks without an overall consideration of the organizations' goals for information needs and information management.

The ten-step outline for the project, based upon the general method for effecting change, was written into the corporation's internal project proposals with the exception of the development of a philosophy statement (step 1) and the use of the survey data to select a system based on needs defined by the users (step 4). Although written into the project outline, the ten-step outline was not followed. Some of the steps were omitted, lowering the involvement of the participants in the design of the project and their subsequent commitment to the project.

Probably the primary factor which probably caused the failure of the project was the compression of the planning phase. In the initial project plan two months were scheduled between the distribution of the survey questionnaires and the beginning of training on the system. The project, as executed, compressed the activities (except the training design) of this two-month period into one week. Had the network and the terminal placement been better planned, the project might have succeeded. The ten-step outline provided interaction with the users in order to fit the system to the network workflow, and, through user feedback, to work out any problems that might arise (steps 5 and 6). But the project started up so rapidly that by the time data had been gathered which indicated problems existed, the usage had dropped to the point that the project had already failed. The study demonstrated that just making a system available will not ensure its use unless there is a deeper consideration of the users' attitudes toward its effectiveness.

The fact that the system was not implemented until a year after it was initially conceived, coupled with the compression of the initial analysis phase, lowered the users' commitment to the system and shortened the time available to discover and remedy the problems experienced in starting it up.

Unfortunately neither the consultants nor the corporation met their objectives for the project. The consultants would have preferred a confirmation of their implementation methodology rather than confirmation of the fact that consideration of user needs and user participation in design and implementation are important to an effective system. The corporation still does not know whether an EMS would be suitable for its 2000 employees. Perhaps the corporation learned what the consultants already knew: a philosophy of "bring in a new system and have the employees use it" is not an effective method for implementing office automation.

REFERENCES

1) Alldredge, R.W. Electronic message systems: Factors affecting their acceptance. Unpublished Masters Thesis, M.I.T., Cambridge, Mass., June 1980.

2) Alter, S. and Ginzberg, M. Managing uncertainty in MIS implementation. Sloan Management Review 20, pp. 23-31, 1978.

3) Argyris, C. Management information systems: The challenge to rationality and emotionality. Management Science 17, 6, 1971, pp. B275-292.

4) Bellinger, W.H. Electronic Mail Systems: Are they effective in the office? Unpublished Masters Thesis, M.I.T., Cambridge, Mass., June 1980.

5) Bjorn-Anderson, N. et al. Eds. The Impact of Systems Change on Organizations. Sijthoff & Noordoff, Germantown, Maryland, 1979.

6) Bramlette, C.A., Jewell, D.O., and Mescon, M.H. Designing for organizational effectiveness. Atlanta Economic Review 27, 5 and 6, 1977.

7) Conrath, D.W. Measuring the computer's impact on organizational structure. In S. Winkler(Ed.) Computer Communication: First International Conference on Computer Communication. Washington, D.C., Oct. 1972, pp. 68-73.

8) Conrath, D.W. and Bair J.H. The computer as an interpersonal communication device: A study of augmentation technology and its apparent impact on organizational communication. Proceedings of the Second International Conference on Computer Communications, Stockholm, Sweden, August 1974.

9) Davis, L.E., and Cherns, A.B. Eds. The Quality of Working Life. The Free Press, New York, N.Y., 1975.

10) French, W.L., and Bell, C.H. Organizational Development, 2nd Ed. Prentice-Hall, Englewood Cliffs, N.J.

11) Ginzberg, M.J. Early diagnosis of MIS implementation failure, promising results and unanswered questions. Management Science 27, 1981, pp. 459-478.

12) Harris, S.E. The acceptance of technology in the office. Journal of Information Management. Fall,1980.

13) Harvey, A. Factors making for implementation success and failure. Management Science 16, 1970, pp. B312-321.

14) Keen, P.G.W. Information systems and organizatioal change. Communications of the ACM 24, Jan., 1981.

15) Keen, P.G.W. and Scott Morton M.S. Decision Support Systems: An Organizational Perspective. Addison-Wesley, Reading, Mass., 1978.

16) Kling, R. The organizational context of user-centered software designs, MIS Quarterly, Dec. 1977, pp. 41-52.

17) Lippitt, M.E., Miller, J.P. and Halamaj. Patterns of use and correlates of adoption of an electronic mail system. Proceedings of the American Institute of Decision Sciences, Nov. 1980.

18) Pasmore, W.A. and Sherwood, J.J. Sociotechnical Systems: A Sourcebook. University Associates, La Jolla, Calif., 1978.

19) Roberts, K.H. and O'Reilly, C.A. Measuring organizational communication. Journal of Applied Psychology 59. 3, 1974, pp. 321-326.

20) Tucker, J.H. A key to office automation: Managing change. Proceedings of the American Institute of Decision Sciences, Nov. 1980.

21) Tucker, J.H. Implementation of decision support systems. Proceedings of the IEEE Conference on Cybernetics and Society, Oct. 1981.

22) Tucker, J.H. The humanmity of office technology, or C.P. Snow in the office of the future. Paper presented at Interface '81, Fifth Annual Humanities and Technology Conference, Oct. 1981.

23) Whisler, T.L. Information Technology and Organizational Change. Wadsworth, Belmont, Calif., 1970.

24) Zand, D.E. and Sorenson, R.E. Theory of change and effective use of management science. Administrative Science Quarterly 20, 4, 1975, pp. 532-545.

25) Zmund, R.W. Individual differences and MIS success: A review of the empirical literature. Management Science 25, 1979, pp. 966-979.

A CASE STUDY OF OFFICE WORKSTATION USE

Christine V. Bullen

Center for Information Systems Research
Massachusetts Institute of Technology, Cambridge Massachusetts 02139

John L. Bennett and Eric D. Carlson

IBM Research Laboratory, San Jose, California 95193

ABSTRACT: This paper describes the use of the Office Analysis Methodology to study a specific office environment in order to determine requirements for an advanced office workstation. The research site environment was unique in providing an opportunity to observe a natural growth pattern in the use of advanced technology. Specific workstation requirements were identified and are being implemented. Interesting observations are reported in the following areas: categories of secretarial work, use of existing workstations, influence of a community of users, access to shared services, and impacts on productivity and organizational behavior.

I. MOTIVATION AND STATEMENT OF PROBLEM

In the Spring of 1981 the IBM San Jose Research Laboratory (SJRL) was faced with an operational problem. A variety of typewriters and terminal equipment, installed in offices throughout the Laboratory, had been acquired over a period of time for use by administrative and secretarial workers. As part of planning for an expansion of physical facilities, a committee began investigating how the equipment was actually used in order to make intelligent recommendations about what new equipment should be provided for the administrative support staff in the future. At the same time, a Computer Science group within the Laboratory was developing an advanced office workstation. It became clear that a study of the work patterns of the administrative support staff in this particular Laboratory could be helpful in understanding the general requirements for an advanced workstation.

After reviewing published methodologies and inviting proposals from academic groups involved with office systems research, the decision was made to work jointly with the MIT Center for Information Systems Research (CISR). CISR was in the process of conducting research into the nature of office work using the Office Analysis Methodology (OAM) developed at MIT [1]. The CISR research is designed to explore office automation issues in several organizations including samples from manufacturing, high technology, and service industries.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0101 $00.75

As Zisman [2] observes, much of the previous work directed toward office automation has focused on mechanization of the current straight-forward, paper-producing tasks. The CISR studies, conducted as part of the CISR research on office automation, encompass the total range of procedures carried out by office workers at all levels – secretarial to top management. The studies seek to identify, through interviews during site visits, procedures that have a high impact on the mission of offices within organizations.

We present here the results of one part of the CISR study conducted at the SJRL. This part focused on existing secretarial tasks performed using existing workstations (Section II). The results (Section III) were used to develop requirements for office workstations for the SJRL (Section IV). Although the results presented here are from a single case study, they do indicate the value of using a systematic methodology such as OAM to study office work. From our study of office environments similar to that found at the SJRL, we believe that the requirements apply generally (Section V) to office workstations in highly automated offices. This paper is a shortened version of a full paper scheduled for publication later this year [3]

II. METHOD

Objectives

The first phase of the specific study at the SJRL had two objectives:

A. Survey the tasks currently performed by the secretarial staff in the SJRL.
B. Understand the current use of the existing workstations and the role that they play in support of these secretarial tasks.

The objectives, narrower than those of other CISR studies, were focused initially in order to answer the shorter range operational question of workstation requirements and selection. The interviews were directed toward understanding the existing procedures and the possible effects arising from the acquisition and installation of new office workstations.

The Office Analysis Methodology

The Office Analysis Methodology (OAM) [1] was used to guide and structure the study. OAM focuses on 1) understanding how each office operates within the organization with respect to the overall organizational

mission, and on 2) understanding how that mission is accomplished. This focus involves conducting a "functional" analysis of the office's operation, expressed in business terms. The procedures being performed and their purposes are identified so that analysts, programmers, and office workers can communicate effectively about requirements.

OAM defines several levels of abstraction as a conceptual framework for gathering data. The MISSION of an office support group (e.g., the secretarial staff) is described in terms of purpose and goal (e.g., support the technical staff by preparing documents, handling phones, and managing office work). A FUNCTION (e.g., document preparation) is the aggregate of all the procedures that INITIATE, MANAGE, and TERMINATE the use of office resources to achieve a business goal (e.g., keying, proof reading, printing for review, and revising text). A RESOURCE is an entity (e.g., a document, a word processor) that is managed to meet a business goal. A PROCEDURE (e.g., an outline of the sequence for printing a photo-composed draft) prescribes the tasks needed to complete an activity. A procedure (or the tasks specified within a procedure) will often involve the manipulation of a specific OBJECT or set of objects. An OBJECT (e.g., a typed page, an instruction book) is a tangible entity that is a component of a resource or that provides information about the resource.

OAM offers potential benefits by avoiding the following pitfalls often encountered in the use of conventional requirements analysis:

- suboptimizing present procedures as a result of a focus on discrete procedures and tasks taken out of context;
- preserving archaic procedures as a result of a concentration on mechanizing a discrete process without gaining an understanding of the bigger picture.

In addition OAM provides the opportunity to identify which activities are valuable in accomplishing the mission, as opposed to identifying only the easily observable, visible, structured tasks. Through this approach, OAM can help to define productivity and isolate meaningful measures that apply to semi-structured tasks found in the office, the work as it is actually carried out [4].

The best available description of OAM is given by Sirbu et al. [1]. We can only give an outline of the concept here. The following illustrates some differences between conventional requirements analysis and OAM:

CONVENTIONAL	OAM
look for processes structured enough to be completely automatable	focus on requirements of functions within the organization (not on operational details)
how much secretarial time is spent typing	how many hours each by how many people does it take to complete a procedure; how often per week is it repeated; how many procedures are in process at any one time
how many forms are filled out per unit of time	how many resources are in process in a unit of time to carry out a business function

The Case Study Site

The IBM San Jose Research Laboratory (SJRL) is organized into four major research areas, called Functions,[1] each of which is headed by a Functional Manager who reports to the Director of the Laboratory: Computer Science, Physical Science, Storage Systems, and Applied Science. All centralized administrative tasks are combined at a Functional level in Administrative and Technical Services (A&TS). The total population – research and support personnel – is about 600 people. Within each research Function there are two to three departments, staffed by Research Staff Members (RSMs), ranging in size from 23 to 45 people. In addition, a number of visiting scientists, postdoctoral fellows, and summer interns report at functional, departmental, or project levels while temporarily at the SJRL.

In this phase of the study, we confined interviews to the secretarial staff working at the Function and Departmental levels. We also included those secretaries working in the Administrative Processing Center giving support to A&TS professionals.

The SJRL provided an intriguing research site. While it is comparable to our other research sites in its basic organizational design and its administrative functions, it is unique in the following ways. First, the SJRL is staffed by technology-oriented, highly skilled professionals who create an environment which is receptive to introduction of new technology. In addition, sophisticated technology is available for use by administrative people in a setting without mandate or formal pressure to employ it. This, combined with the supportive access to information from RSMs, (who use computer facilities in their routine paper drafting, presentation preparation, and messages to colleagues), results in an unusual situation for studying natural growth patterns in the use of advanced technology.

We used the OAM framework to construct a one hour interview. The categories of those interviewed and the range of people nominally served by each secretary in the category are as follows:[2]

Level	Number of Secretaries Interviewed	Number of People Served by Each Secretary (Range)
Functional Manager	4	3 to 7
Department Manager	9	23 to 43
Administrative/Other	8	5 to 45
TOTAL	21	

[1]The use of the term "function" is not identical to the use of the term in OAM. However, both uses relate to a focus on activities needed to achieve mission results.

[2]The selection of those interviewed and the range estimate for the number of people served were made from an inspection of the organization chart.

Each secretary was interviewed at the place of work where sample objects (computer-readable and paper copy) could be displayed as needed to serve as an illustration. All secretaries interviewed used a desk-top display terminal capable of showing simultaneously on the screen 24 lines of 80 uppercase and lower case characters. Some had a terminal allowing display of 43 lines at a time. Each terminal had an attached (but movable) keyboard. In addition, each secretary or administrative support person had a communicating typewriter terminal for printing output on letterhead paper. This terminal was also used occasionally as a stand-alone typewriter, and it had magnetic card storage.

The display terminals, which had no stand-alone data entry capability, were attached to a large-scale host computer operating the VM/CMS system. Also attached to the computer, directly and through a network, were many high speed and/or high quality devices used to produce printed output on a variety of paper and pre-printed forms. The network is an IBM corporate network linking computers in most IBM Laboratories worldwide. The secretaries in the sample were using a variety of software available on the system, including a full-screen editor, a document formatting and printing facility, a message system, and a number of locally-developed macro programs.

III. RESULTS FROM THE INTERVIEWS

Types of Tasks

In our study we focused on resources and objects that resulted in paper copy or that went through a keyboard data-entry phase. We did not address phone handling as a task (except to note approximate percentages of time spent), although we did consider typed lists as support for making phone calls.

The secretarial work at SJRL can be divided into the two categories as follows:

1. Work initiated by "others": secretary is
   a. told explicitly what to do
      (given raw text and a sample letter specifying the format)
   b. given some discretion
      (foil format)

2. Work initiated by the secretaries in response to:
   a. being told in a general sense to achieve a result
      (produce an equipment inventory list)
   b. being told they are responsible for a result
      (making labels used to forward mail to former visitors)
   c. observing a need and taking responsibility for meeting it
      (monitoring department expenditures)

The "others" initiating secretarial work are professionals, managers, and visitors serving on the staff. The category of "work initiated by others" is the one that generally comes to mind when people describe what the "typical" (actually stereotypical) secretary does. The work is text-oriented – that is, the initiator provides text (handwritten, dictated, rough-typed), and the secretary's role is to provide text output in typewritten or printed form. Completion requires little contact with other resources (documents or people). The tasks require a fixed format which is made standard through policy, tradition, or equipment constraints. The outputs are typically a file specifically designed to be revisable (because the final task result is subject to initiator negotiation) and text-on-paper for initiator review. Examples are notices, letters, and memos (relatively short) and activity reports (relatively long).

Hiltz and Turoff [4] describe the category of "work initiated by others" well when they observe that secretaries "act as intermediaries between the originators and the recipients of text." They comment on the fact that word processors are typically aimed for one specialized aspect of what the secretary actually does.

In the second category, the levels of self-initiated work reflect increased taking of responsibility and creativity. Although this category involved keyboard data entry, the data entry is not an end in itself. Completion generally requires contact with others, and the secretary has flexibility in selecting the format in which the results are presented. The outputs are typically files used by the secretary in carrying out office procedures. Examples are mail logs and reminder files. There seem to be a series of prerequisites for secretarial work in the "self-initiated" category to begin appearing. First, powerful tools (or a light work load) must make it possible to get routine work out of the way. Then the secretary must have a willingness to explore the use of tools in imaginative ways. Third, the professionals served must acknowledge the value of the resulting innovation.

A major result of our interviews is the observation of how time is allocated between categories of work. The secretaries to Function Managers spend less than 50% of their time[3] doing structured text entry initiated by others. Secretaries to Department Managers spend from 50% to 90% of their time doing such work, depending on the style of the Department and the style of the individual secretary. The remaining secretaries spend 75% to 95% of their time on this category of tasks.

The Nature of the Documents

Because document preparation, storage, retrieval, and printing are major office workstation tasks, we used the OAM "objects" concept to investigate how documents enter into the SJRL secretarial work flow. We categorized documents on the basis of frequency as seen by the secretary (Figure 1). The study identified over 50 different (in terms of format) documents, about equally divided among the three categories. Document preparation can be either self-initiated or initiated by others, but it is low on the scale of value-added. However, document preparation may be a task associated with a much more significant procedure (e.g., planning for and ordering new office equipment) that may be important to the mission of an office.

[3] The time estimates were collected in the interviews and reflect the judgment of those interviewed. The figures were not independently validated. However, the results are consistent within the hierarchy of secretaries, suggesting that these approximations are reasonable. The interview results were supplemented by some direct observation.

A SAMPLE OF DOCUMENTS
AND THEIR FREQUENCY OF USE

High Activity	Medium Activity	Less Frequent
travel expense authorization	invention disclosure	work authorization order
purchase order	presentation clearance	award
check request	request for publication	patent disclosure
petty cash	foreign travel approval	verification of foreign seminar attendance
time card	personnel data	
travel plan	an order for manuals	
Manuscript Processing Center work order		

Figure 1. Documents are frequently completed by the secretary in order to support the staff. Additional secretarial time is spent tracking progress and checking for completion.

The Use of Existing Workstations

Each secretary had access to the display terminal that was connected to the host computer and to the communicating typewriter terminal with printer and magnetic card storage. We were interested in the secretarial preferences for choice of use and the criteria entering into the decision (Figure 2). We expected that these preferences would illustrate characteristics of work style or features deemed useful by the secretaries.

CRITERIA FOR USE OF DISPLAY TERMINAL
AND TYPEWRITER TERMINAL

- 90% preferred use of display terminal at all times unless:
  - observed slow response time due to heavy host usage
  - needed special letterhead on short notice
  - observed host service was temporarily unavailable
- 100% preferred typewriter terminal for:
  - filling out forms
  - short, one-time jobs (envelopes, local, occasional mem
- Confidential or sensitive text
  - 14% preferred using typewriter terminal start to finis
  - 48% preferred input of text on the display terminal, printing on the typewriter terminal (host-connected)
  - 38% did not do this kind of work

Figure 2. The secretarial staff made a choice in use of the display terminal and the typewriter terminal.

To explore this further, we asked the secretarial staff for their opinions about what they liked when using the display terminal. These characteristics are listed in Figure 8. Note that these are really opinions about the display terminal and the services accessed via these terminals. The secretarial responses can be summarized: assuming the host system was delivering normal service, the display terminal was always preferred except when there were "printing" constraints (e.g., a special letterhead was required, the task consisted of addressing a single envelope, or the text was personal and confidential). We also asked the secretarial staff for their opinions about what they liked when using the display terminal. In general, the responses here described characteristics of the display terminal (e.g., ease of manipulation and correction of text) and features of the services accessed via the terminal (e.g., communications features).

These opinions reflect two issues – productivity enhancement and "expectations" for text output quality. The secretaries unanimously felt that the ease with which text could be manipulated and changed on the display terminal was a major aid in supporting their ability to "get the work out the door". The message and text transmission features saved time and footsteps. They also provided support communications among secretaries and staff which were never before possible when RSMs were working at home or at other IBM locations.

As a result of their extensive use of the display terminal and host-based featured, the secretaries also made a number of observations of things they would like to see changed. A first and foremost concern was the lack of formal training. As a result, they suspected the system contained a number of additional features that they would find valuable, but they had no easy way to confirm this. Other concerns addressed included logistics of access to remote printers, host unavailability and slow response time, and physical size and layout of terminal equipment.

The Community of Users

The research staff (RSMs) in the SJRL make extensive use of the VM/CMS system and the display terminals for administrative as well as technical support. The RSMs in the Computer Science Function, in particular, have an understanding of how the underlying system is organized, have a professional interest in adding new capability to the system, and have a personal interest in seeing this new function used. These combine to act as a powerful influence for innovation in how secretarial and administrative work is done.

For example, individual staff members will develop a new way to use the computer (e.g., a SHIP macro command for transferring files between local users and over a network). They will tell or show colleagues how the new command makes their job easier. Some colleagues will typically discover additional things that could be done and will add the function to the system. Since staff members are accustomed to doing much of their own secretarial work (due to provision of tools to support this[4]), several will discover ways that they can use the system for office activities. Staff members, in the course of enlisting secretarial support for a task, explain how to use the system to get that particular job done. A secretary, typically receptive to a new idea, will make some notes on how to use that feature. Then at lunch, or in the course of an exchange with another secretary, the technique will be passed on. The readiness to accept innovation, coupled with an easily used message system, leads to the spread of such information over the informal network.

[4]Investigation of staff member work is part of Phase 2 of this study and is not documented here.

Note that this process can operate two ways. Ideas that seem at first to be good, may upon wider use, be discovered to have flaws. For example, a RECEIVE macro command allowed incoming files to overwrite user files that happened to have the same name. The word soon spread to "watch out", and many began using a similar but accident-proof RD (for "read") macro command instead. Thus, the SJRL environment tends to separate the unworkable from the workable.

Shared Services

There is a variety of services available on the host computer which are accessed by the secretaries through the terminal. These include an online telephone directory, a program to check the spelling of words in text documents, hand calculator functions, and the capability to send and receive "soft copy" documents and messages.

The secretaries consider the computer-based message system to be a significant aid in their work. They use the facility to exchange brief notes, to seek help on special problems in system use, and to exchange techniques by sending special formats to one another. There appears to be a significant use of this in preference to use of the telephone. While the telephone is an instrument for direct and immediate conversation (once the connection is made), the staff is aware of the disruptive nature of a phone call. By using the message system, the exchange can be managed by the recipient on a "when convenient" basis, whereas conventional office etiquette requires that a ringing phone be answered. Computer messages can be duly noted without interrupting the flow of the work in progress.

IV. REQUIREMENTS FOR OFFICE WORKSTATIONS

The primary objective for this part of the CISR/OAM study at the SJRL was to help determine the requirements for administrative workstations. As previously mentioned, all secretaries in the study had "dual" workstations (display terminal and communicating, magnetic-card typewriters). This "dual workstation" configuration was expensive and consumed considerable space (about 15 cubic feet, requiring an entire desk top surface area). In addition, much of the equipment was reaching the end of its useful lifetime, and many terminals had noise and quality problems. Thus, the SJRL administrative management was interested in replacing these workstations. The management, and many secretaries, were aware of CRT word processors and were interested in their possibilities for use at the SJRL.

The results of the first part of the CISR/OAM study described here were used to make recommendations on office workstations for the SJRL and to guide research on office workstations. As we summarize the requirements derived from the study, we will interpret the study results which led to each requirement (Figure 3).

Two of the study results strongly indicated that an office workstation should conveniently attach to the host (VM/CMS) system. The attachment should be high speed (at least equal to the 1.2 megabit rate of the display terminals), should involve a simple hardware and software protocol, should provide emulation of the display terminal, and should permit file transfer between host system applications and the workstation. First, the study indicated a high utilization of, and dependency of secretaries upon, host system services (e.g., messages, data bases, editors). Any

RESULTS	RELATED REQUIREMENT
1a. Dependency on host services. b. Community of users.	1. Host attachment with terminal emulation and file transfer.
2. Magnetic card typewriter used for short documents, backup, and printing.	2a. Typewriter emulation. b. High quality printer.
3. Dual workstations used at the same time.	3. Multi-tasking with simple, quick user support for switching between local and host-attached modes.
4. Wide variety of documents.	4. Support for wide variety of formats with prototype templates.
5. Screen, availability, printer, and training problems.	5. Larger screen; local processing, storage; and training programs.

Figure 3. The results from the study indicate specific requirements for an administrative workstation.

workstation which does not provide access to these services would reduce productivity and incur substantial resistance. To assure transfer of established work patterns, access to the host system should provide the same user interface and functionality as the display terminals, at least as a subset. Second, the study indicated the importance of the community of users. As observed, this community provides training, help, and new applications. If the workstation does not provide access to the shared services, the value of this community of users will be lost.

Adequate host attachment would allow an office workstation to provide the functions of the existing display terminals, but the workstation also needs to provide the functionality of the typewriter workstations. In particular, the study indicated that the typewriter workstations are used for printing, storage of classified, short or personal documents, and for preparation of documents that are short or difficult to prepare with the display terminal (e.g., forms). Support for document preparation would have to include "typewriter emulation" for short documents, such as envelopes. One of the advantages of the current typewriter workstation is that it can be used in a standalone mode. Many of the secretaries noted this advantage when asked if they could give up their typewriter workstation. In addition, several secretaries noted that the word processors which they had seen did not provide this typewriter capability.

As previously discussed, the secretaries work with a wide variety of documents. Many of these are prepared, stored, printed, and retrieved using the existing workstations. Thus, any new workstation must support these tasks and documents (e.g., letters, memos, numerous forms, research reports, and lists). The file transfer capability of the host attachment must be integrated with the local support for these documents so that a document that is prepared at the workstation can be processed by the appropriate host applications. For example, it should be possible to format

and print documents prepared at a local workstation with existing host applications. It should also be possible to store in host data bases the forms which are filled-in at a workstation, and it should be possible to use the stored content as data inputs to existing accounting applications. A workstation that could support more of the current documents than the existing workstations would be a definite productivity aid, particularly if that support included prototype forms (partially filled-in) serving as templates.

Finally, the problems with the existing workstations that the secretaries listed indicate requirements for a new workstation. Larger screens, smaller packaging, local processing and storage, high-quality printer, and training programs would be considered advantages.

The study results thus indicated both an opportunity and a set of requirements for an administrative workstation. Based on this set of requirements and a review of existing workstations, we were unable to identify an existing product that met the requirements.

As a result, the administrative management decided not to replace the existing workstations at this time. In addition, a project in the Computer Science Department has begun to develop extensions to currently available IBM products in order to provide a workstation that meets the requirements.

V. DISCUSSION

Although the observations and list of workstation requirements which came out of this study are not necessarily surprising, they are important. The results of the study were the major inputs into the decision to continue using the current workstations and into the enhancements to be made to the IBM Displaywriter. In addition, the study provided a number of useful insights on administrative functions and the potential for office automation at the SJRL. We conclude with a discussion of these insights.

We were able to identify missions, functions, procedures, and tasks in offices through use of OAM. The results included classes of work considered both typical and less typical in the secretarial world. The SJRL environment provided a good opportunity to observe a natural growth pattern in the use of advanced technology. This gave us an opportunity to "see the future", to get a glimpse of the potential for office automation. Our observation of the workstation features which are used in performing advanced office procedures gives us confidence that our list of workstation requirements is responsive to future secretarial needs.

A number of interesting organizational and behavioral implications of office automation were identified. Perhaps the most significant is the importance of a community of users. Another interesting observation is in the area of user resistance. At the SJRL we did not see a reluctance to use new technology. Obviously the high-technology environment has a significant impact on creating an atmosphere for acceptance. In addition, the secretarial self-selection process plays a role. There appear to be at least two additional factors which contributed to low resistance. First, the heavy work load resulting from the ratio of support staff to professional staff creates a situation where secretaries are highly motivated to enhance their own ability to get the work done. It is clear to all of them at the SJRL that without the aid supplied through the host system, they would be hopelessly inundated with work. In addition, because the professional staff makes use of the computer-based tools themselves, the early "rough draft" versions of paperwork are seldom directly typed into the system by the secretaries.

The fact that virtually everyone at SJRL has a terminal and that many RSMs do their own text entry raises the issue of professional staff resistance to personal use of keyboards. Although this issue was not directly investigated in this study, some observations were made as a result of the interviews. The professional staff is enthusiastic and positive about the computer support for self entry of documents directly on the system. In at least some of the instances where resistance was indicated, it seemed to be based on a sensitivity or insecurity related to the job in general, not to the computer technology. This is an unusual and no doubt controversial observation. Ideas analogous to this have been put forth by Turkle [6] as she reported the perceptions of data processing professionals when they described their own relationship with computers.

In this part of the study, we did not focus on productivity measures. However, secretaries stated they have found the time to organize tasks and create procedures for smoothing work flow which they could not even think about before. This has been stimulated both by time saved through computer support and by access to the tools powerful enough to create these procedures. Some tasks in this category include automated distribution lists, reminder files triggered by a clock, and financial monitoring and analysis systems. Another stated advantage of the existing office automation tools was the ability of the secretaries to take on new tasks, many of which are self-initiated. Here it is not a case of doing "more" of the current tasks or of doing them "faster", but rather the presence of entirely new functions. The most significant example is the message system. The secretaries learned to use the message system to inform each other when they would be temporarily away from their desks or to arrange for one secretary to do tasks (e.g., pick up printed documents) on behalf of several secretaries.

The results of an OAM study generally include more detail on procedures and their relationships to the mission of the organization. Such detail can be used to suggest potential or desirable organizational impacts which can be expected from office automation. Our initial use of the OAM was for gathering information related to the objectives outlined in Section II; thus, we did not go into detail on issues surrounding organizational impacts. However, some observations in this area did result from the interviews. First, the use of terminals by professionals to input their draft documents is an impact of this technology which clearly affects professional work. Secondly, the ability of the secretary to take on more self-initiated tasks in support of the office mission affects the nature of secretarial work. Finally, the ability of anyone to send electronic messages to anyone, unfettered by traditional office etiquette, creates the potential for a significant simplification of office communications.

REFERENCES

1. M. Sirbu, S. Schoichet, J. Kunin and M. Hammer, "OAM: An Office Analysis Methodology," Laboratory for Computer Science, OAM-16, October 1980.

2. M. Zisman, "Office Automation: Revolution or Evolution?" Sloan Management Review, Spring 1978, pp. 1-16.

3. C. Bullen, J. Bennett, and E. Carlson, "A Case Study of Office Workstation Use", IBM Systems Journal, 1982, in press.

4. E. Wynn and L. Suchman, "Procedures and Problems in the Office Environment," XEROX, Advanced Systems Department, Palo Alto, April 1979, 18 pages.

5. S. Hiltz and M. Turoff, "The Evolution of User Behavior in a Computerized Conferencing System," Communications of the ACM, Vol. 24, No. 11, November 1981, pp. 739-51.

6. S. Turkle, "Computer as Rorschach," Society, January-February 1980, pp. 15-24.

INTEGRATING DIVERSE KNOWLEDGE SOURCES IN TEXT RECOGNITION

Sargur N. Srihari, Jonathan J. Hull and Ramesh Choudhari

Department of Computer Science
State University of New York at Buffalo
Amherst, New York 14226

ABSTRACT

The capabilities of present commercial machines for producing correct text by recognizing words in print, handwriting and speech are very limited. For example, most optical character recognition [OCR] machines are limited to a few fonts of machine print, or text that is handprinted under certain constraints; any deviation from these constraints will produce highly garbled text. This paper describes an algorithm for text recognition/correction that effectively merges a bottom-up refinement process that is based on the utilization of transitional probabilities and letter confusion probabilities, known as the Viterbi algorithm [VA], together with a top-down process based on searching a trie structure representation of a lexicon. The algorithm is applicable to text containing an arbitrary number of character substitution errors such as that produced by OCR machines.

The VA is a method of finding the word that maximizes likelihood over all possible letter combinations and not necessarily those in a lexicon; it is based on a dynamic programming formulation which leads to a recursive algorithm. The algorithm can be viewed as a shortest path algorithm through a directed graph called a trellis. The negative of the log transitional probabilities are associated with the edges of the trellis and the negative log confusion probabilities are associated with the nodes. The cost of a path is then the sum of all the edge and node values in the path. Some generalizations of the VA have used either a fixed but small number of alternatives per letter of the input word, called the Modified Viterbi Algorithm [MVA] or a variable number of alternatives per letter; these alternatives can be determined by the letters that have the highest confusion probability.

The VA (and its variations) is a purely bottom-up approach whose performance may be unacceptable due to the fact that the resulting strings do not necessarily belong to a dictionary. One approach to this problem is to use top-down contextual information, in the form of a dictionary of allowable input words, to aid the bottom-up performance of the VA.

The trie data structure is suitable for determining whether a given string is an initial substring, or prefix, of a word in the lexicon. Essentially, the trie considers words as ordered lists of characters, elements of which are represented as nodes in a binary tree. Since the VA proceeds by computing for a given length the most likely prefix, the trie is an efficient data structure.

Simultaneous search of the VA trellis using a variable number of alternatives per input letter and the trie structure is achieved using a binary array A. Element A[j,i] is set to 1 if the jth letter of the alphabet is a possible correction for the ith letter of the input word, and 0 otherwise. Thus the paths of the trellis that need to be evaluated are only those that begin at the 1's of the first column of A and proceed through the 1's of the subsequent columns. Before evaluating a path that proceeds from one column of A to the next column, that path is determined to be legal with respect to the trie. The computational complexity of the resulting algorithm is of the same order as the VA.

Experimental results with the algorithm are described for a data-base of 6372 words of garbled English text using a lexicon of 1724 words represented as a trie of 6197 nodes. With first-order letter transitional probabilities and the optimum number of alternatives per letter (with respect to execution time) the

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

algorithm was able to correct 87% of the garbled words. This is in contrast to correction rate of 35% with a purely bottom-up approach and 82% with a purely top-down approach. As an example, the garbled sentence

"If we looi at what has produsedlomputer imtelligence qo far, we see multiple lamers, each of which rests on primitives of the naxd tayfr dowm, forminc a hierarcfical structure with a great deal interposed between the intelligent prphvem and the transistors which ultimatelu suppodt it"

is corrected by the algorithm to

"If we look at what has produced -------- intelligence so far, we see multiple layers, each of which rests on primitives of the next sites down, forming a hierarchical structure with a great deal interposed between the intelligent program and the transistors which ultimately support is",where the hyphens indicate a rejected word. Rejections can be reduced (or eliminated) by increasing the number of alternatives per letter.

Thus the algorithm is able to effectively utilize top-down knowledge in the form of a lexicon of legal words, channel characteristics in the form of a table of letter confusion probabilities and two-types of bottom-up information: letter shapes represented as vectors and letter transitional probabilities.

Message Files

D. Tsichritzis
S. Christodoulakis

Computer Systems Research Group
University of Toronto
Toronto, M5S 1A1, Canada

1. Introduction

Consider an office information system and the messages which circulate within it. We will assume for the purposes of this paper that the messages are text messages in machine readable form. We are interested in designing a message filing capability which has the following characteristics:

1) It can deal with a wide variety of messages
2) It can retrieve the messages in a flexible manner
3) It provides a simple and uniform interface to the user
4) It can be implemented efficiently for a large volume of messages

We define messages as consisting of a *header* and a *body* [Kirstein]. The header contains formatted data representing the most important characteristics of the messages, e.g., sender, date, destination, etc. The body is text consisting of a series of words. We will denote by $A_0, A_1, \cdots, A_n$ the attributes of the header. A_0 is a special attribute which contains a unique system-wide identifier for the message. The body will be denoted as an attribute B of type text. A particular message will be represented as $(a_0, a_1, \cdots, a_n, b)$. All messages do not have to be of the same type. Each type of message, however, is represented by a set of attributes and a body. In the two extreme cases we have the message type (A_0, B), i.e., documents and $(A_0, A_1, \cdots, A_n)$, i.e., records. Forms as messages can be represented as $(A_0, \cdots, A_n, B)$ with the additional stipulation that the values of A's are dispersed within B.

2. Message storage and retrieval

People in offices find paper messages by filing them appropriately. A *message file* is a labelled container of messages which groups messages according to their meaning. For instance, messages from a particular person about a particular project are filed together. Notice that a message file in the office environment allows filing of different types of messages within it (unlike a computer file). To help pinpointing the right file name for retrieval, file names can be structured according to their relationships. A common structure is a hierarchical structure as provided, for instance, by folders within drawers of file cabinets. People search a message file by scanning sequentially to obtain the appropriate message(s). They are guided by a partial specification of contents of the message and are assisted by a certain vague image of what the message looks like.

There are some characteristics of office message files which we would like to retain for electronic filing. However, there are also limitations which we can and should avoid. First, we observe that it would not be difficult to duplicate the same kind of manual message filing in terms of electronic messages. For example, UNIX files provide hierarchical directories for file names [Ritchie and Thompson]. Within each file messages can be stored sequentially as byte strings. We only need to implement a sequential scan of messages which takes a sequence of bytes and displays it as a message. We would expect to find such capabilities associated with any kind of file server as part of an office information system. However, there are some limitations in such a filing facility.

First, the structure of the files is seldom static. The world changes, the message characteristics change and their filing method should change accordingly. Even if we have achieved a perfect filing method encapsulated in a directory structure, it will soon be out of date. Reorganization of the files is difficult even in a computer environment. Messages will have to be reassigned to different files in a new directory. Second, many messages can potentially refer to many subjects and should be associated with many files. In such a case, we either have to duplicate the messages (with associated consistency problems) or we need to have pointers to messages rather than the messages in many files. Finally, no matter what file organization we choose we will not be able to divide easily messages in small groups according to contents. As a result, if the volume of the messages becomes high, the files will be large and sequential scan will be time consuming.

We should augment the filing capability with an access method which can retrieve messages according to contents. In this way messages can be organized in general files rather than complex directories, e.g., 1981 general correspondence, project X memos, etc. The information retrieval capability enables the user to retrieve quickly the desired message from the file. Since the messages are filed in very general terms, frequent reorganization of files is also not needed.

It would be nice to retain the properties of sequential scan. That is, messages are still retrieved one at a time simulating a sequential scan. It is, however, a "lucky" sequential scan. The user specifies a *filter*. Mes-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0110 $00.75

sages which do not qualify according to the filter are skipped. Messages which qualify are displayed. It is also important that they are displayed in a format recognizable to the user. In this way, the user can visually check the message before he accepts it.

In this paper we will concentrate on the specification, analysis, and implementation of such a filtering capability. We will assume that all messages are stored in large general files according to user and role. The specified filter restricts the attention to a manageable subset of the messages. Messages within the subset are obtained sequentially for a final check by the user.

A major thesis of the paper is that the filtering capability does not have to be exact. We assume that the user will seldom be able to specify an absolutely tight filter. His specification in terms of contents will allow more messages to qualify in addition to the ones he absolutely wants. His visual inspection will finally pinpoint the desired messages. If the specification of the filter is not exact, its implementation does not have to be exact. In other words, suppose the specification of the filter allows 10% non-relevant messages. If the implementation of the filter adds another 0.5% of non-relevant messages, the user will not care. The system could eliminate this extra 0.5% with some additional processing. However, it is as easy for the user to deal with 10.50% as to deal with 10% of unwanted messages. Thus it may be desirable to pass the results of the query to the user and not to a program.

3. File organization

Given a specification of the filter in suitable internal form we need a file organization which restricts our attention to the set of appropriate messages.

The specification of values for A_i's can make use of standard indexes, e.g., B-trees for their evaluation. The specification and evaluation of a pattern is more difficult. We could use indexing methods for words but we will need much space to store the word indexes [Haskin]. In addition merging the pointers obtained from the indexes is a time consuming problem [Haskin]. We propose instead a method of reducing the sequential search to a manageable level. The organization of the messages is sequential. The search is still sequential but we search an auxiliary file rather than the message file.

Consider a word W appearing in a message. We will denote by $S(W)$ a bit string which represents (not uniquely) the word W. There are many different algorithms for transforming W into $S(W)$. One method is to partition W into triplets or quadruplets of letters and to hash each partition of letters into a fixed number of bits. The bits are concatenated (up to a maximum length) to form the signature $S(W)$. A fixed number of bits may preceed the signature $S(W)$ to indicate its length. Alternatively, signatures of words may be of fixed length.

Consider a set of messages of the form $m=(a_0,a_1,\cdots,b)$. Suppose that we physically file all the messages sequentially as they come into a general physical file F. Let i_k be a pointer to a message within F. That is the file system will deliver the message on the basis of i_k. We will also construct signature files which will aid the search for the message.

When the user stores a message he will specify one (or more) logical files in which he wants the message filed. For each logical file f there will be an associated physical file f_s which stores not the message but a representation of the message which we will call the *signature* of the message. The signature of the message will consist of $(i_k,t,S(a_0),\cdots S(a_n),S(b))$ where:

i_k is a pointer to where the message is stored

t is the type of the message

$S(a_0),\cdots,S(a_n)$ are exact representations of $a_0,\cdots,a_n$

$S(b)$ is the signature of the body.

The signature of the body is a sequence of bits which approximately represents the significant words of the message's body. This is obtained by the following algorithm. Traverse sequentially the body b. For each word check whether it is one of a set of common words (the, that, a, ...). If it is set $S(W_i)=\varphi$, i.e., skip the word. If it is not substitute the word W with its signature $S(W)$. Skip also the blanks and formatting symbols.

The sequence of words comprising b is represented by the bit string $S(b)$ in the following manner.

If a word W is in b then $S(W)$ will be a substring of $S(b)$ provided that W is not a common word. Notice that the opposite is not true. That is, $S(W)$ may match a substring of $S(b)$ without W be a part of b. This situation may arise for two reasons. First, two words W and W' may have the same signature $S(W)=S(W')$. We may, therefore, find W' rather than W in b. Second, if we use variable length word signatures without a prefix that specifies the length of the signature, the bitstring $S(W)$ may match the suffix and prefix of the signature of words $W'W''$. This situation arises because blanks are eliminated.

Consider a series of messages $\{m_1,\cdots,m_t\}$ with bodies $\{b_1,\cdots,b_t\}$ and a word W. The set $\{m_i \mid S(W)$ matches $S(b_i)\}$ is a superset of the set $\{m_j \mid W$ matches $b_j\}$. It is important that the difference between the cardinality of the two sets is small.

4. Access method

A user specifies a query to the message filing facility by first indicating the logical file f and the type of message t he is accessing. The system displays the appropriate template for t. The user optionally specifies some values for $A_0,A_1,\cdots,A_n$ and a pattern P by partially filling the template. Values entered for $A_0,A_1,\cdots A_n$ are interpreted as selection conditions. The pattern P can be a complicated expression of words involving arbitrary boolean combinations of regular expressions of words. Messages that satisfy the query are these that satisfy the conjunction of all selections specified, and the pattern P of words appears in the body of the message.

To find the set of qualifying messages the system scans sequentially the signature file of S associated with f. For each message signature it checks first the type. If the type does not match it skips the signature. If it does it tries to match first the given values for the attributes and then the signature of the body.

Consider the pattern P as expressed by a regular expression of words. The signature of the pattern $S(P)$ is a regular expression which is obtained by substituting $S(W)$ for each significant word in P retaining the regular expression operators, dropping common words and substituting $\{0\cup 1\}^*$ for each unconstrained interval of words. Matching message bodies for P is substituted with matching the signature of P, $S(P)$ against the signature of the bodies $S(b)$. The signatures $S(b)$ are binary strings and $S(P)$ is a regular expression of binary strings. Testing $S(b)$ for $S(P)$ is equivalent with testing whether

$$S(b)\,\varepsilon\,\{0\cup 1\}^*\,S(P)\{0\cup 1\}^*$$

This is a recognition problem for finite automata [Aho, Hopcroft, Ullman]. It can be shown that for a specific regular expression R the membership of a string in R can be checked by a NFA with number of states at most double the number of symbols in R [Aho, Hopcroft, Ull-

man]. It follows that given a pattern P testing for $S(P)$ in the bit strings $S(b)$ can be done by a NFA with states a linear function of the symbols in the pattern P.

5. Implementation

We are implementing a message filing facility as discussed in this paper based on UNIX. We are going to integrate this facility into a message management system.

We envision the facility in three different environments.

In the first environment we assume a workstation running UNIX, e.g., ONYX, Z-lab, LSI 11/24, with a small 20 Mb disk. Suppose that we can devote up to 10Mb of the disk for message filing. We feel that we should be able to store of the order of 10,000 messages of less then a page length on the average. The messages will be organized in approximately 20 different logical files which will hold about 500 letters each. The signature of such logical files will be 50 Kb which can be searched quite fast even by a workstation processor.

In the second environment we assume a message file server operating on a high power microprocessor, e.g., M68000 based system like APOLLO with a 300 Mb disk. This facility can serve approximately 20 users connected to the server via a local network. Each user can store on the average 10,000 messages with similar organization and response as in the first environment. The total capacity of the message file server is of the order of 200,000 messages which can probably be adequate for a department or groups of persons.

In the third environment we envision a high power microprocessor with a 300 Mb disk storing only the signatures. In such an environment we can go up to 2,000,000 messages. The messages themselves are stored on microfilm using an addressable reader printer, e.g., KODAK. In such an environment the user specifies his request over a network and gets the associated messages using a telefax facility. Such an environment will probably be well suited for organizational archives of messages.

Notice that in all three cases the software for searching for the messages is the same. We can see, therefore, a nice combination of facilities or a natural transition from one environment to the other.

We should also mention that if some messages are not in machine readable form they can still be read using OCR which is probably centralized. In such a case messages can go as they came, on say microfilm, while their signatures are produced and retained for searching purposes.

Finally, we should point out that since the problem of document retrieval is reduced to membership problem of regular expression the whole technique can be cast into VLSI [Floyd and Ullman].

References

Aho, Hopcroft, Ullman: *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

Aho Kernigham and Weiberger: Awk-A Pattern Scanning and Processing Language (Second Edition), Bell Labs, Murray Hill, New Jersey, Sept. 1978.

Christodoulakis a: Estimating Selectivities in Data Bases, PhD Thesis, Technical Report CSRG #136, University of Toronto, 1981.

Floyd and Ullman: The Compilation of Regular Expressions into Integrated Circuits, *Proc. 21st Symposium on Foundations of Computer Science*, Oct. 1980, pp. 260-269.

Haskin: Special Purpose Processors for Text Retrieval, *Database Engineering 4*, 1, Sept. 1981.

Kirstein: New Text and Message Services, *Proc. IFIP Congress 1980*, Tokyo, Melbourne.

Ritchie and Thompson: The UNIX Time-Sharing System, *CACM 17*, 7, July 1974.

Sevcik: Data Base System Performance Prediction Using an Analytic Model, *Proc. VLDB 1981*, pp. 182-198.

Shannon: Prediction and Entropy of Printed English, *Bell System Tech. J. 30*, 1, 1951, pp. 50-64.

Tsichritzis: Integrating Data Base and Message Systems, submitted for publication, 1981.

Tsichritzis: An Office Form System, *Proc. VLDB 1980*, Montreal.

Tsichritzis and Lochovsky: *Data Models*, Prentice Hall, 1981.

A VOICE RESPONSE SYSTEM FOR AN OFFICE INFORMATION SYSTEM

Dik Lun Lee

Computer Systems Research Group, University of Toronto

Toronto, Canada. M5S 1A7

ABSTRACT

One of the major activities in offices is communication. Communication involves information exchange in different kinds of media such as text, voice, pictures, and most likely, a mix of these. As such, a successful office information system must provide adequate facilities for handling these kinds of communication media. This paper discusses the design and implementation of a voice response system for OFS (Office Form System), a prototype office information system.

1. INTRODUCTION

Office automation is prompted by a need to increase the productivity of office workers, thereby reducing the cost of operating an office. It involves a number of computer science disciplines such as software engineering, database management, computer architecture, communication, and human factors [Ellis, 1980]. The challenge is to apply the results obtained in these disciplines to an office information system. After some years of research, several major problem areas have gradually been identified [Tsichritzis, 1982] [Ellis, 1980] [Tsichritzis and Lochovsky, 1980].

First, facilities for automating office work need to be integrated. Traditionally, such facilities include database management systems, electronic mail, word processors, telephone, facsimile, typewriters, photocopiers, etc. Unfortunately, these facilities are often provided to users in separate pieces, usually supplied by different manufacturers with different areas of expertise. The problem confronting office information systems is how to integrate such a large variety of facilities into a single system while keeping them in a uniform environment with a clean user interface.

Second, office procedures must be automated, rather than merely mechanized. To this end, users must be able to specify office procedures to the system with a powerful, yet easily understandable, specification language. Once specified to the system, appropriate actions can be initiated by the system without the intervention of humans when certain conditions are satisfied.

Third, tools must be provided to represent the structure of the office, to specify and to analyse information requirements and information flow within

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0113 $00.75

an office, and to predict the effects of changes in office organization. Such tools will enable managers to restructure the office systematically to suit the new information era. In addition, administrative staff can use these tools to design and verify office procedures. Examples of office modeling tools are Office Form Flow Model [Ladd, 1980] and Information Control Nets [Ellis, 1980].

As a subproblem of the first area (i.e., integration of facilities), the integration of communication facilities has particular importance. Historically, the use of computers in the office has been for data processing (e.g., database management systems, acounting, and payroll preparation). However, the major activity in an office is not only data processing but, with equal importance, interpersonal communication, whether formal or informal. Communication involves information exchange in different kinds of media such as text, voice and pictures, and most likely, a mix of these media. As such, a successful office information system must be able to deal with both *formatted* data (e.g., data represented as records) and *unformatted* data (e.g., text, voice and pictures).

The emphasis of this paper is on incorporating speech communication in an office information system. Specifically, a facility for synthesizing speech from text is discussed and its design and implementation within OFS (Office Form System [Cheung, 1980]), a prototype office information system, are described.

2. OFS (OFFICE FORM SYSTEM)

OFS is a prototype office information system designed and developed at the University of Toronto. A brief overview of its purpose and function are given below. A discussion of the concept of form management can be found in [Tsichritzis, 1982]. Details about OFS can be found in [Cheung, 1980] [Tsichritzis, 1980].

A form can be used as a concept for integrating various facilities and services in an office information system. It can be thought of as a logical image of a business paper form which is used in almost every office, but it is more general, flexible and elaborate than an ordinary paper form. OFS is a computerized system for handling electronic forms. In OFS, forms can be created, retrieved, copied, printed, mailed, traced and put into a dossier just as can be done with ordinary business paper forms. However, there are some facilities in OFS (existing or capable of being incorporated in the future) which have particular importance in an office information system.

OFS is implemented on a network of workstations on which users work. As such, forms can be mailed from one station to another without (observable) delay. Global querying is possible in that information about forms residing in other workstations can be obtained. OFS is compatible with a database management system (Micro Relational System, MRS [Kornatowski, 1979] [Hudyma, 1981]) so that information about forms can be queried using facilities provided by the database management system and by OFS. In the former case, relational commands are available to experienced and authorized users to operate on a collection of forms. This allows information about a number of forms to be obtained and processed. In the latter case, forms are processed one at a time. Forms can be accessed sequentially or by specifying conditions on attribute values.

In the general case, a form can be viewed as a message in that its contents convey information.

Information exchange is actually carried out when a form is mailed from one station to another. Traditionally, a message is regarded as unformatted and it is up to the recipient to interpret the message. In an electronic mail system, a message is often defined as a header and a body [Kirstein, 1980]. However, this is not enough to allow flexible message processing facilities, such as mail classification, for two reasons [Tsichritzis, 1981]. First, the header structure is uniform for all messages, so it can not be embedded with adequate information to allow the system to interpret a large variety of messages. Second, the header structure can not evolve with time.

On the other hand, a form imposes structure on a message while maintaining great flexibility (i.e., a form can be redesigned and a user can add as many form types to the system as he thinks appropriate). Furthermore, the structure of a form guarantees a unique interpretion of its contents which would be otherwise ambiguous, if not meaningless. For instance, the attribute value "1980" can stand for a year, an amount or simply a serial number. But if this string appears in a form with label "publication year", then it stands for a year; the message conveyed to the user is clearly defined.

In OFS, attribute values are extracted from a form and stored separately as a relation tuple. When a user retrieves a form, the tuple is retrieved and filled into a *template* through which structure and meaning are assigned to the attribute values. For instance, a text template can be defined to map attribute values to form a text document. Figure 1 shows a text template for a book form. Attributes are numbered from left to right, top to bottom, starting with 1 for the form key. They are referenced by their numbers.

FORM KEY:_____

BOOK FORM

CALL NUMBER:_____

TITLE:_____

AUTHOR:_____

PUBLICATION YEAR:____ PRICE:_____

ENTERED ON:_____ STATION ID:_____

Figure. 1

The separation of attribute values and templates allows flexibility in processing a form. For one thing, the attribute values, stored internally as tuples, can be accessed by a database management system. For another, the same set of attribute values can be structured in different ways and be interpreted differently by the system by using different templates. In this regard, a template resembles a schema and subschema in a database management system.

Apart from defining the spatial structure of a form, information can be embedded into a template so that the system knows how to manipulate the attribute values as they are mapped to the output device. Depending on the template, attributes can be mapped onto a display terminal, a speech synthesizer, a graphics terminal, a printer and so on. Hence, templates can be distinguished according to their function as *text* templates, *voice* templates, *graphic* templates, *printer* templates and the like. In fact a mix of these templates can be used if a form uses multi-media output (i.e., combining text, pictures and voice messages in one form). The following is an elaboration of some of the facilities that can be provided by templates.

Since a template can be thought of as a general mapping, attribute values can be modified as they are mapped to a template. An attribute can be attached with a procedure which takes the attribute and, optionally, other attributes as input and produces an output to the template. The final output can be different from what is actually stored in the system. For instance, the full name of a person can be stored in the system but a template can map only the last name, the last name with initials or map a different form of the name to the output. Furthermore, the mapping can be intelligent enough to determine the title of that person from other attribute values such as marital status, sex and education qualification.

Since all attributes are represented internally as a bit string, it is necessary to tell the system how to interpret the bit string. Consider a graphics template. An attribute can be a character string, in which case, the bits are interpreted in groups of eight. Further, different fonts can be used when the attributes are displayed. On the other hand, an attribute can be a picture. In this case, the coordinates, size and resolution of the picture must be defined to the system. In addition, the encoding scheme of the picture must be known. It can be a bit pattern of the picture. It can also be compressed with a certain algorithm or simply be a set of graphics instructions which define the picture. All this information can be defined within a template so that as long as a user is provided with a suitable template, he does not have to worry about the internal representation of the attributes. This has an implication that communication between heterogeneous systems is possible if the template(s) is sent along with its attributes to the destination. This is very important in an office information system.

Imposing structure on messages means that the user is not left free to enter any message on a template. He must enter data in the way defined by the structure of the template. This could be a constraint which sacrifices flexibility but it is a way to avoid ambiguity and guarantee consistent interpretation of messages. Furthermore, the constraint can be useful in some situations. For instance, if a field is an utterance, a voice template can define the way in which the utterance is stored (e.g., using digitization, LPC or formant synthesis, or text-to-speech synthesis [Schafer, 1975] [Makhoul, 1975], [Flanagan, 1970], [Allen, 1976]) and the rate of output. In text-to-speech synthesis, linguistic ambiguities would arise. Imposing a structure on the text can help to resolve some of the ambiguities.

3. VOICE TEMPLATES

In a voice template, as in a text template, attributes are numbered from left to right and from top to bottom. If an attribute value is to be spoken out, it must have a corresponding entry in the voice template. An entry consists of five parts: a number, a prefix utterance, a trailing utterance, a flag and a dictionary. Since a text-to-speech synthesizer is used in the prototype system, attribute values which are to be spoken out, the prefix, and trailing utterances are actually stored as character strings.

If an entry is numbered by n, it is associated with the nth attribute value in a relation tuple. If an attribute, n, is to be spoken out, an entry, numbered n, must be found in the voice template.

The prefix and trailing utterances provide a similar facility as "fill in the slot" in a text template in that an attribute value can be prefixed or appended

with other utterances as it is spoken out. The prefix utterance serves to qualify the attribute, (e.g., "amount is", "customer name is", etc.). The trailing utterance serves primarily as the unit of the attribute value, like "dollar".

The dictionary associated with each attribute serves to resolve some linguistic ambiguity and exceptions not handled by the synthesizer. Words separated by blanks, commas or a period are stripped out of the attribute value and are matched with words in the dictionary. If a match is found, the word is substituted with the corresponding word or phrase defined in the dictionary. Using a separate dictionary for each attribute implies that substitution of a word can be done according to the context in which it appears. This helps to resolve some ambiguities easily. For instance, the word "mrs" can be the abbreviation for "misses" or the acronym for "Micro Relational System". The form administrator, who designs the voice template, can determine the correct context and specify the appropriate substitution in the dictionary associated with the attribute (e.g., "misses" should be used if "mrs" appears in a name field such as "customer name" whereas "MRS" should be used if "mrs" appears in a item field such as "product name".

The problem of homographs (i.e., the same word such as the word "record" having a different pronunciation when it is in different word classes) can also be solved, to a certain extent, in the same way. If a field of a form is so strongly "typed" that the form administrator can be sure that a word (e.g. "record") would appear in this field only as a verb (or noun), then a dictionary entry can be specified to make the appropriate substitution to reflect the stress pattern. Without making use of prior knowledge about the characteristics of an attribute value, these linguistic ambiguities must be resolved by performing complex syntax and semantic analysis to determine the word class of each word [Cherry, 1978]. Of course, there are some situations in which little information can be known beforehand. For instance, a "comment" field would be so unconstrained that no prior knowledge about its contents can be derived to resolve any ambiguity. If correct speech output is desired, a procedure which can analyse the syntax and semantics of natural language should be associated with this field. In any case, a great saving in processing time can still be achieved since speech output of this kind of field would not be frequent.

In some situations, brute force substitution is not effective, if at all possible. For fields whose input stands for an amount, the string of numbers should first be converted to speech forms, (e.g., "123" should be converted to "one hundred twenty three"). Another example is the "date" field in which "sat" should be converted to "saturday". This kind of conversion is done by specifying a conversion procedure for the field that needs conversion. As such, a *flag* field is provided to qualify the attribute value so as to invoke the appropriate procedure. Examples of flags are:

Y (for year) — which causes a number to be spoken as a year in the conventional way. For instance, "2000" is translated to "two thousand"; "1980" is translated to "nineteen eighty".

D (for date) — which expands a date format to its full text. For instance, "Thu Oct 1 1980" is translated into "Thursday October first nineteen eighty".

A (for amount) which causes a number to be spoken out as a quantity. For instance, "1980" is spoken as "one thousand nine hundred and eighty".

If no flag is specified, a number will be read out digit by digit. Other procedures can also be attached to an attribute by specifying other flags if the form administrator feels that it is necessary.

Utterances can be defined in a voice template such that they are spoken before any attribute is spoken or after all attributes are spoken. These can be used, for instance, for informing users of the form type of the form being processed or for greeting or alerting messages.

In a voice template, an entry which corresponds to an attribute value has the following general format:

```
attribute_number prefix_utterance,trailing_utterance,flag
    dictionary_entry : substitution
      .....    : ....
      .....    : ....
```

Figure 2 is a voice template defined for the book form given in figure 1.

```
-2 book form
2 call number
3 title is
        MRS   : M R S
        OFS   : office form system
4 authored by
5 published in,,Y
6 price is,dollars,A
7 entered on,,D
-1 form end
```

Figure. 2

The sentence labelled "-2" will be spoken before any other utterance. The sentence "-1" will be spoken after all attributes have been output. Negative values are used to distinguish them from other entries which correspond to attribute values of the form. Since no entry is labelled by "1" or "8", the form key and the station id will not be spoken out. If no voice template is defined for a form type, then no voice output will result when a form of that type is displayed. The rest of the template is defined as described above. Like text templates, voice templates can be created and modified with an editor. They are managed by the form administrator. Ordinary users are not allowed to modify them.

4. IMPLEMENTATION ISSUES

OFS is written in the programming language C [Kernighan, 1978] and runs under the UNIX operating system [Thompson, 1978] on a LSI 11/23. The voice response system is also implemented in the same environment.

The synthesizer used in the system is a text-to-speech synthesizer from Votrax [Votrax, 1980]. It has a RS-232 serial interface and accepts characters as input. As such, it can be treated by the computer as a terminal. In the current implementation, a TTY driver is used for interfacing the synthesizer. The synthesizer also has an internal buffer for holding input characters. Two modes of operation are available. First, it interprets the characters in the buffer as English text and uses a text-to-speech algorithm to translate the text into speech directly. Second, it interprets the characters as phonemes and as such operates as a phonetics synthesizer. It is possible to switch between these two modes under software control by sending a sequence of control characters.

The text-to-speech algorithm can translate 85% of the words correctly. To handle the exceptions, such as words adapted from foreign languages,

proper nouns, etc., another dictionary is used. Every word must be matched with this dictionary before it is output to the synthesizer. If a match is found, the corresponding substitution is made. To distinguish between this dictionary and the dictionaries defined in a voice template, it is called the *global dictionary* while dictionaries defined in a voice template are called *local dictionaries*. They are different from each other. First, a local dictionary applies to a single field of the corresponding form while the global dictionary applies to all words which are to be output to the synthesizer. Second, in local dictionaries, the substitutions are specified as English text (i.e., they are translated by the same text-to-speech algorithm). In the global dictionary, phonemes are used in the substitution. The reason for using English text in a voice template is to make the voice templates independent of the synthesizer used since the phonemes used by different synthesizers can be different. The form administrator can assume that a perfect synthesizer is available to him when he defines voice templates. The global dictionary is used to supplement the synthesizer so that it looks like a perfect one.

Options are provided in OFS so that a user can enable or disable the voice output at any time during a work session. This is important since the usefulness of voice output depends on the application. For instance, an experienced user may find the voice output disturbing while a novice user or a user who is too busy to look at the screen may find the voice output very useful. An option is also provided to users so that the output of the current form to the synthesizer can be repeated.

When voice output is enabled, an attribute value which has a corresponding entry in the voice template is processed as follows:

1) The flag of the attribute is checked. If a flag is found, the attribute value is processed by the procedure specified by the flag and steps (4) and (5) are executed next.
2) If a flag is not found, the words in the attribute value are stripped out and matched to its local dictionary. If a match is found, the word is substituted accordingly.
3) The processed attribute value is then concatenated with the prefix and trailing utterances defined in the voice template to form a new string.
4) The words in the resulting string are stripped out again and matched to the global dictionary. If a match is found, a word is substituted by the corresponding phonemes together with control characters which switch the synthesizer from text to phoneme mode and switch it back to text mode after the phonemes are output.
5) The processed string is output to the synthesizer.

From usage, it has been found that the speech output for this system is quite intelligible. Since the attribute values of a form are usually very short, the intonation is not very important.

5. CONCLUSION

Speech offers humans a means of spontaneous and effective communication. Information exchange can be effected as soon as a message is formulated in the brain; no tool, no preparation is required. In an office, speech is even more important for communication between people. As such, an office information system must provide adequate facilities for handling speech. Those facilities include voice editing, voice

filing, voice electronic mail, and voice synthesis and recognition [Maxemchuk, 1980], [Nacon, 1980], [Koekebakker, 1981]. An important requirement is the capability of handling voice and other media such as text and pictures in an integrated way. These facilities are important in reducing communication cost and increasing human acceptance of an office information system. The help of a voice response system to blind people is obvious.

In an office environment, no fixed pattern of verbal communication can be captured a priori. Thus, the system must be able to synthesize speech from unrestricted text. As such, a text-to-speech synthesizer is used in our prototype system. The voice response system enables any field of a form to be spoken out in a meaningful way by using a voice template. An attribute is qualified with a prefix utterance such as "amount is" and followed by another one such as "dollars" when it is output from the synthesizer. Furthermore, the way to speak an attribute can be modified according to context. For instance, "1978" is translated into "nineteen seventy eight" when representing a year and "one thousand nine hundred and seventy eight" when representing an amount. The system also enables voice messages, as opposed to attributes, to be embedded in a form. The linguistic ambiguities inherent in text-to-speech synthesis are handled by associating each field with a dictionary which is defined in the voice template.

Text-to-speech synthesis is the most economic way for implementing an unlimited vocabulary voice response system. The most serious drawback is its low speech quality. However, research effort has been directed to developing rules to translate text to speech with prosodic features (e.g., stress patterns) [Smith, 1981]. With the low cost of memory and high processing power of microprocessors, such a system can soon be expected to be implemented on a microprocessor.

6. REFERENCES

[1] J. Allen, "Synthesis of Speech from Unrestricted Text," Proceedings, IEEE, Vol. 64, No. 4, Apr. 1976, pp. 433-442.

[2] C.K. Cheung, J. Kornatowski, "The OFS Users' Guide," Computer Systems Research Group, University of Toronto, 1980.

[3] L.L. Cherry, "PARTS - A System for Assigning Word Classes to English Text," Computer Science Technical Report #81, Bell Lab., Murray Hill, N.J., 1978.

[4] C.A. Ellis, G.J. Nutt, "Computer Science and Office Information Systems," Computing Surveys, Vol. 12, No. 1, Mar. 1980, pp. 26-60.

[5] J.L. Flanagan, et al., "Synthetic Voices for Computers," IEEE, Spectrum, Vol. 7, No. 10, Oct. 1970, pp. 22-45.

[6] R. Hudyma, et al., "MRS: A Microcomputer Database Management System," Proceedings, ACM Workshop on Small Database Systems, Orlando, Florida, Oct. 1981, pp. 174-180.

[7] B. Kernighan, D. Ritchie, "The C Programming Language," Prentice-Hall, Englewood Cliffs, N.J., 1978, pp. 521-536.

[8] P. Kirstein, "New Text and Message Services," Proceedings, IFIP Congress, Tokyo, Melbourne, 1980, pp. 521-535.

[9] J. Koekebakker, "Computer Voice Response/Recognition," Canadian Datasystems, May 1981, pp. 26-31.

[10] J.Z. Kornatowski, "The MRS Users' Manual," Computer Systems Research Group, University of Toronto, 1979.

[11] I. Ladd, D. Tsichritzis, "An Office Form Flow Model," Proceedings, NCC, May 1980, pp. 533-540.

[12] N.F. Maxemchuk, "An Experimental Speech Storage and Editing Facility," The Bell System Technical Journal, Vol. 59, No. 8, Oct. 1980, pp. 1383-1395.

[13] J. Makhoul, "Linear Prediction: A Tutorial Review," Proceedings, IEEE, Vol. 63, No. 4, Apr. 1975, pp. 561-580.

[14] R.J. Nacon, "Voice Electronic Mail for the Business Executive," Proceedings, NEC, 1980, p. 233.

[15] R.W. Schafer, L.R. Rabiner, "Digital Representations of Speech Signals," Proceedings IEEE, Vol. 63, No. 4, Apr. 1975, pp. 662-677.

[16] W. Smith, "Phonemes, Allopones, and LPC team to Synthesize Speech," Electronic Design, June 1981, pp. 121-127.

[17] K. Thompson, D. Ritchie, "The UNIX Time Sharing System," Bell System Technical Journal, Vol. 57, No. 6, Part 2, 1978, pp. 1905-1929.

[18] D. Tsichritzis, "OFS: An Integrated Form Management System," Proceedings, 6th International Conference on Very Large Data Bases, 1980, pp. 161-166.

[19] D. Tsichritzis, "Integrating Data Base and Message Systems," Proceedings, 7th International Conference on Very Large Data Bases, 1981, pp. 356-362.

[20] D. Tsichritzis, "Form Management," CACM (to appear), 1982.

[21] D. Tsichritzis, F.H. Lochovsky, "Office Information Systems: Challenge for the 80's," Proceedings IEEE, Vol. 68, No. 9, 1980, pp. 1054-1059.

[22] Votrax, "Votrax Model SVA Synthesizer Interface Specification," Votrax Division, Federal Screw Works, Michigan, 1980.

SESSION 8

VIEWS OF OFFICE DATA

Chairperson: S. Peter deJong
Massachusetts Institute of Technology

Panel Members

Richard E. Fikes
Xerox PARC

Carl Hewitt
MIT

Vania Joloboss
CII - Honeywell

Lance A. Miller
IBM

Office applications will eventually be defined and executed by office personnel rather than professional programmers. There are many scientific disciplines and technologies that are contributing to this goal. This panel will explore a few of these areas: databases, artificial intelligence, and linguistics. Today the active database research areas are in highly distributed, real-time transaction oriented systems, and dynamic expressive data organizations such as the relational model. The extending of the database models to cover the requirements of the office is a major concern in office systems research. Artificial Intelligence has been directly involved with the problems of interfacing with end-users for a considerable period of time. The AI knowledge base systems are much closer in spirit to the way humans accumulate and use partial knowledge to solve problems than the database models. Using AI techniques, expert systems have been built for various application areas. One can envision the use of expert office systems to help in solving business and organizational problems. Much of the communications done in the office is in terms of natural language. Linguistics is concerned with the structure of language and how people generate it and understand it. The greater the extent that an office system can deal with natural languages, the more useful that system will be to end-users.

This panel will discuss how intelligent the next generation of office systems can become using these technologies.

SEDL -- A LANGUAGE FOR SPECIFYING INTEGRITY CONSTRAINTS ON OFFICE FORMS

James C. Ferrans

Western Electric Company
Lisle, Illinois 60532

Abstract

ODIN is a system developed by Western Electric to automate the construction of electronic form entry, processing, and retrieval systems. An important component of ODIN is SEDL, the language used to define the logical structure of forms and the integrity constraints that each form must pass as it is entered.

SEDL is a powerful, concise end-user language. Its integrity constraints are sophisticated assertions which are combined by using standard programming language constructs. Lists on forms are considered to be relations in a miniature relational database, and data manipulation language constructs are used to specify constraints involving them. SEDL's expressive power has been more than adequate for the complex office applications that have used it.

Introduction

This paper describes integrity constraint specification in the SEDL form definition language. SEDL is part of the ODIN application generation system, and has been in successful use for two years.

The paper is organized as follows: In this introduction I will discuss the motive for developing an office form integrity constraint language tailored for end users, and present some themes connecting the features of SEDL. Then, I will discuss the logical structure of SEDL forms. This leads into a description of the central application data dictionary, which contains all the information relating to a set of forms. Next, I introduce the SEDL sublanguage used to define abstract field types, and follow it with a description of the form definition sublanguage. I conclude with a summary of what we have learned from the SEDL development, and some directions for future work.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0123 $00.75

Integrity constraints should be written in a language designed especially for the task rather than in a general purpose programming language. The special purpose language provides constructs geared to integrity constraint specification, and hides much of the implementation detail. Thus it is more concise than a programming language, and closer to the user's thought processes.

However, the constraint language must be sufficiently powerful. If it is not, the users must still provide error check modules in the programming language or resign themselves to a higher rate of undetected errors. The language should also be geared to end users: the ones who formulate constraints are not usually programmers, and if they can write in this language they can dispense with programming support and the attendant communication overhead.

The reader should notice several themes running through this language description. First, there is a philosophy of error detection embedded in the language, namely that errors are to be detected and reported as soon as possible in the interactive form entry process, subject to the restriction that this should not confuse the data entry personnel.

Next, note how conciseness is achieved through SEDL's extended set of comparison operators and through its mechanisms for handling repeating groups of fields. Finally, notice the self-documenting features. English keywords are used wherever possible, even to the point where comparison operators have English synonyms.

Form Structure

SEDL views a form as an ordered set of contiguous fields (see Figure 1). There are no limits to the way that a form can appear to the user: the data entry program may only allow forms that fit on one screen, or it could allow forms that fit on multiple successive screens, or it could act as if the screen were a window onto an arbitrarily large form. There is no need to enter forms via CRT at all. They could be entered on a hardcopy terminal, or in bulk mode from a data file. A data entry program might even allow the user to access information in a database through a combination of several forms at once.

One or more of the fields must serve to uniquely identify a particular form among all forms of the same type. These fields are known

PATIENT INFORMATION RECORD

Name: _____ SSN: _____
Street: _____
City: _____ State: __ Zip: ____

Recent Office Visits

Date	Reason	Fee	Test Fee	Total
__/__/__	_____	_____	_____	_____
__/__/__	_____	_____	_____	_____
__/__/__	_____	_____	_____	_____
__/__/__	_____	_____	_____	_____
__/__/__	_____	_____	_____	_____
__/__/__	_____	_____	_____	_____
	Totals:	_____	_____	_____

Figure 1. Sample patient information form.

as the form's key. In the sample form, the key is the social security number.

Data is entered and stored in human readable format, so each field holds a character string of fixed maximum length. While some data items are truly character strings (e.g., "name" and "city"), others are integers ("day") or real numbers ("fee"). Other fields could be binary, octal, or hexadecimal integers.

Some fields are naturally grouped together to form larger objects called structures. For example, "month", "day", and "year" are normally thought of collectively as a "date". Other fields, or groups of fields, are repeated on a form. For instance, the form in Figure 1 has a repeating group of patient visit descriptions.

Much thought was given to how these repeating groups should be viewed and manipulated. Initially, they were considered arrays and were to be manipulated with normal programming language constructs (do-loops, indices, and temporary variables). However, this proved to be less than satisfactory because it violated the goal of keeping SEDL as high-level and non-procedural as possible. The user should not have to say how to make a check, just what check to make.

The solution was to consider the form as a tiny relational database. In the relational data model, as introduced by Codd [Codd70], a database consists of a set of tables (called relations) with rows (called tuples) and columns (called attributes). Each attribute takes its values from some underlying set of values called a domain. For example, the attribute "day" would draw its values from the domain of all integers from 1 to 31. Given that all attributes in a database are not repeating groups of smaller objects (i.e., given that the database is in first normal form), Codd showed that there was a powerful set of mathematical operations on relations. These operations are the basis of several high-level non-procedural data manipulation languages (DMLs) which can be used to specify complex integrity constraints.

In SEDL, each repeating group, or list, is considered a relation. Its rows are called elements, and correspond to tuples in the relational data model. Similarly, its columns are called attributes. Simple fields can be thought of as relations having one tuple and one attribute, while structures have one tuple and several attributes. By viewing a form as a tiny relational database, SEDL could build on the existing work on specifying integrity constraints in relational database management systems (e.g., [Hamm76] & [Ston75]), and perhaps influence this work in return.

However, the end-user should not be burdened by being forced to think of simple fields as relations. When checks do not involve lists, people think in terms of simple logical and arithmetic expressions, if-then-else statements, and the like. The approach was thus to develop a set of more or less standard programming language constructs for checking simple data items, and merge in relational DML constructs geared to formulating integrity constraints on small databases. An analogous approach has been advocated in the domain of general purpose programming languages by Prenner and Rowe [Pren78], who proposed a programming language containing relational data manipulation constructs as a way to combine the power of a DML with the flexibility of a programming language. The DML chosen to use as a starting point in SEDL was SQL [Cham76], since it used English keywords and also formed the basis for the ODIN multi-form integrity constraint language.

The form structure chosen, and the work already done on integrity constraint languages, led us to implement four distinct classes of error checks on individual forms. These classes are:

- domain checks - checks made on individual field values immediately after they are entered.
- structure checks - checks made on the fields of a structure immediately after they have all been entered.
- list element checks - checks made on the fields of a particular list elem-

ent immediately after they have all been entered.

form checks - checks made on the entire form once it has been entered.

Any additional categories of integrity constraints would confuse the data entry personnel without adding to the language's power.

The Data Dictionary

Each data item in an application is an abstract representation of an object or concept that has meaning to that application. For example, a social security number represents a person. In form entry applications different data items will often represent the same type of object or concept: a payroll form could have an employee's social security number on it, and an insurance form could have the social security numbers of the employee and his beneficiaries.

Because of this commonality, a SEDL user first defines the abstract types of data items that will appear on the application's forms. These definitions are called domains, and include data transformation rules as well as a description of the values in the domain. The domain definitions are written in the domain specification sublanguage of SEDL, and are compiled and placed in a data dictionary.

The next step is to define the forms in the form definition sublanguage. A form is a list of fields, each of which is logically connected to an underlying domain. The form specifications are compiled, and in the process the domains are looked up in the dictionary. The form compiler then places a definition of the form in the data dictionary for use by other ODIN tools. This approach not only reduces the overall form development effort, but it imposes a valuable discipline on the user.

The central data dictionary concept is common to many database management systems. McLeod [McLe76] suggested that domains be defined in a high-level language that includes a description of the domain, the ordering rule, and the action to take when a value violates the domain constraint.

Specifying Domains

Domains can either be simple domains or structure domains. For a simple domain, the user must provide its name, type, and length. The type is the "natural" domain from which the SEDL domain draws its values. For instance, the domain of valid human ages is a subset of the natural domain of all integers. The length is the maximum width of the domain's values on the display screen. If the domain is a subset of the real numbers, it must also be given a precision, which is the number of decimal digits allowed. The user may elect to specify a domain check. If he does not, it defaults to one for values in the natural domain.

Ignoring explicit domain checks for the moment, some examples of the simple domains used to build the patient information form of Figure 1 are:

```
zip:    integer(5);

fee:    real(7,2);      /* length 7, precision 2 */

state:  character(2);
```

To define a structure domain, the user gives the names of the attributes in the structure:

```
date:
    structure ( month, day, year );
```

The domain of a structure attribute is assumed to have the same name as the attribute. If this is not the case, the domain must be specified explicitly:

```
flight:
    structure
    ( from domain(airport),   to domain(airport) );
```

Specifying Forms

There are two parts to a form specification. The first is a mandatory description of the form's logical structure, and the second is an optional definition of any list element checks and form checks to perform. Figure 2 is a form specification for the example form, without integrity constraints.

The specification is a list of the form's fields in data entry order. Key fields are entered first, and then the non-key or data fields. Notice that the distinction between attribute and domain pays off well on this form; six fields (attributes) share the "fee" domain. Notice also how the office visit list is specified. Its name is followed by the number of elements and a list of its attributes.

Once the patient form and its domains have been defined like this, the user could go on and create the visual image of the form and use it for data entry. The only integrity constraints applied would be the default ones on the natural domains. The next step is to supply explicit domain checks.

Domain Checks

Domain level integrity constraints fall into two major classes: simple domain checks and structure domain checks. Structure domain checks will be covered in the next section.

To make an explicit check on a simple domain, the user provides four things, each of which has a reasonable default. The first is a values clause, which gives the set of all legitimate values. Next, there is an error message, which will be displayed whenever a value is entered that is not in this set. Next is a suspect values clause, which delineates the set of all values that are legal, but rarely occur. Finally, there is a warning message that explains to the user why a value is suspicious:

```
patient: form;

    key_field: { ssn };

    data_fields:
      {
        name,
        street,
        city,
        state,
        zip,

        visit_list[6]
          {
            date,
            reason,
            fee,
            test_fee   domain(fee),
            total      domain(fee)
          },

        total_fee        domain(fee),
        total_test_fee   domain(fee),
        grand_total      domain(fee)
      };


    list_element_checks:
        /* list element checks are specified here */

    form_checks:
        /* form checks are specified here */

end form;
```

Figure 2. Partial form specification.

```
fee:
    real(7,2)
    values:          { 0 thru 9999.99 }
    error:           "fee must be positive!"
    suspect_values:  { 0 thru 1.00 }
    warning:         "we don't accept charity cases!";
```

Each values clause denotes a set of values. These sets can simply be enumerated:

```
{ "AK", "AL", "AR", "AZ", "CA", "CO", "CT", ... }
```

but there are a number of ways to describe sets as well. For instance, the "fee" domain was described with the thru keyword. If fees had to be rounded to the nearest fifty cents, this could be specified with the thru and by keywords:

```
{ 0 thru 9999.50 by .50 }
```

There are several predefined sets of values. For example, a value can be checked to see if it is alphabetic:

```
{ alphabetic }
```

Single values and sets of values may be combined to form a values clause:

```
{ -17, 0, 11 thru 17, 20 thru 42 by 2, 100 }

{ "1", "2", alphabetic }
```

When a set of values cannot be described with the above mechanisms, regular expressions can be used. For example, if a part number is an "A", "B", "C", "D", or an "X" followed by a dash, one to three digits, and a second dash and two more digits (e.g., C-322-47), the following regular expression will describe the domain:

```
{ regex("^[A-DX][-][0-9]{1,3}[-][0-9]{2}$") }
```

This approach to pattern matching in high-level languages is advocated by Ditzel [Ditz78], and in our experience it is not too difficult for end-users.

Structure Checks

A structure is just a small form that cannot contain lists or other structures. Thus structure level integrity constraints are a subset of the form level constraints and will not be discussed other than to give an example:

```
flight:
  structure { from, to }

  structure_checks:
    from != to
      error: "flight takes off and lands at the same place!";
  end checks;
```

This constraint is invoked whenever the data entry clerk fills in a "flight" structure, and it asserts that the aircraft is not scheduled to take off and land at the same airport.

Simple Form Checks

This section describes those form checks that do not deal with lists.

Integrity constraints are assertions on the values in the form. If an assertion fails, some action must be taken. Thus SEDL is built around the assertion statement, which is a logical expression followed by an action to take if the expression evaluates to false. The possible actions are (1) to issue an error message and (2) to issue a warning.

Logical expressions have a full complement of simple conditional operators (<, >, <=, >=, =, !=), logical operators (and, or, not), and arithmetic operators (+, -, *, /, **, modulus). Parentheses are used to alter or clarify operator precedence. In addition, SEDL has substring and concatenation operators, the length built-in function, and an extended set of conditional operators. Here are

a few simple assertions:

```
grand_total = total_fee + total_test_fee
   error: "patient's fees do not balance";

code >= 6 and code <= 12
   error: "code must be between 6 and 12";

phone.area_code = "312"
   warning: "employee should live in 312 area code";
```

The left-hand side of a logical expression may be compared with sets of values:

```
code between 6 and 12
   error: "code must be between 6 and 12";

part_number != regex("^[A-C][0-9][2]")
   error: "part# cannot start with an A, B, or C and two digits!";
```

and again, as in domain checks, several values or sets of values can form larger sets:

```
class_code not_in { 6, 9 thru 18 by 3, 117, 121 }
   warning: "obsolescent class code!";

part_number[1,1] in { "A", "B", "X" }
   error: "part number must start with an A, B, or X";

part_number in { numeric, alphabetic }
   error: "part number must be all digits or all letters";
```

Assertions can be conditionally made with the **if-then-else** and **case** statements, and combined with **do-end** blocks:

```
if city = "Chicago"
   then do;
      state = "IL"
         error: "Chicago is in Illinois!";
      zip between 60600 and 60699
         error: "illegal Chicago zip code";
   end;


case ( month );
   when { 4, 6, 9, 11 }:
         day <= 30
            error: "bad date";
   when { 2 }:
         if ( year mod 4 = 0 and year mod 100 != 0 )
           or ( year mod 400 = 0 )
            then day <= 29
               error: "bad leap year date";
            else day <= 28
               error: "bad non-leap year date";
   otherwise:
         day <= 31
            error: "bad date";
end;
```

Finally, English synonyms for comparison operators are provided to increase readability:

```
if level is "elementary"
   then grade must be in { "K", "1", "2", "3", "4", "5", "6" }
         error: "bad elementary school grade";
```

List Element Checks

List element checks are a sequence of **on** statements. Each **on** statement specifies the assertions to make on each element of a particular list immediately after that element is entered:

```
on visit_list element entry:
   total must equal fee + test_fee
      error: "the total fee doesn't balance";
```

If more than one assertion is to be made on each list element, a **do-end** block can be used.

Form Checks Involving Lists

Integrity constraints on lists have to have a more complex structure than those involving only simple fields. The user must be able to look at all or part of a list, combine lists, build lists from simple fields and expressions, and use parts or properties of lists in logical and arithmetic expressions. In SEDL, this is accomplished by using constructs derived from relational data manipulation languages.

The For Statement

The **for** statement bridges the gap between the standard programming language constructs in SEDL and the DML constructs. Its purpose is to iterate through the elements of a list. In this respect it is like the **on** statement, but it can be nested in other statements and statements inside it do not have to reference only the fields of a particular element:

```
if city = "Evanston"
   then for every visit_list element:
         total >= 20
            warning: "people in Evanston can afford more";
```

Many of the simpler list checks can be formulated using the **for** and **on** statements, so some users will not have to learn the DML portions of SEDL.

List Constants

To specify more complex list checks it is important to distinguish between expressions that are list-valued and expressions that are scalar-(i.e. simple) valued. Until now all expressions have returned a single value as their result. List expressions return an entire list as their result.

The simplest list expressions are list constants; here are some examples:

```
null

{ <1903>, <1927>, <1952> }

{ <"red", 150, "John">, <"brown", 125, "Mary"> }
```

The first constant is the null list, which has no elements and an indeterminate number of attributes. The next constant has three elements and one attribute, while the last has two elements and

three attributes.

List Specifiers

To reference lists on the form itself, one uses a list specifier. A list specifier says what list or lists to look at, and which rows and columns to pick out from the list(s). The result is another list that can be used in subsequent operations. To illustrate the effects of some list specifiers, assume that a form has two lists on it and that the following values have been entered into the lists:

inventory[5] { part#, in_stock, due, price }

part#	in_stock	due	price
125	10	30	4.50
341	0	100	12.00
999	0	20	5.00
114	100	0	10.00
331	40	0	3.00

orders[5] { customer#, part#, quantity }

customer#	part#	quantity
1923	114	10
1923	444	150
1246	114	200
1431	444	110

The following examples show four list specifiers and the lists they produce when operating on the above lists.

Example 1:

{ select * from orders }

Result 1:

customer#	part#	quantity
1923	114	10
1923	444	150
1246	114	200
1431	444	110

This list specifier says to look at all attributes and elements of the "orders" list. The **select** clause names the attributes to pick out (the "*" is a shorthand for "all attributes"), and the **from** clause tells what list to pick the attributes from. An optional **where** clause tells which elements to pick out; if it is omitted, as above, all elements are chosen.

Example 2:

{ select unique part# from orders where quantity > 100 }

Result 2:

part#
444
114

This selects only the "part#" attribute, and only from those elements of "orders" where the "quantity" is more than 100. The **unique** keyword causes duplicates to be discarded from the resulting list.

Example 3:

{ select part#, due from inventory where in_stock is 0 }

Result 3:

part#	due
341	100
999	20

This shows two attributes being selected.

Example 4:

{ select customer#, orders.part#, quantity, price
from inventory, orders
where inventory.part# equals orders.part# }

Result 4:

customer#	orders.part#	quantity	price
1923	114	10	10.00
1246	114	200	10.00

This example involves the (conceptual) formation of a relational cross-product of two lists. Here, the list specifier will take the two lists and look at every conceivable pairing of an "inventory" element with an "orders" element. Each pair forms a single element in a new list called the cross-product of "inventory" with "orders". This cross-product will have 4 + 3 = 7 attributes and 5 * 4 = 20 elements. The **where** clause then proceeds to pick out rows of the cross-product, and the **select** clause picks out columns. Notice that because there are two different "part#" attributes in the cross-product, the references to them had to be qualified.

Union, Intersect, and Minus

Lists having similar attributes can be combined by using the set operations **intersect**, **union**, and **minus**:

{ <18>, <21> } union { select age from age_list }

{ select part# from orders }
intersect
{ select part# from inventory }

List expressions have no meaning in themselves; they are merely a way of referencing data. The next section describes how they are used to form assertions.

Integrity Constraints Involving Lists

SEDL provides several operators on lists whose results are scalar values. These are **average**, which computes the average of a list column; **sum**, which sums a column; **min** and **max**,

to find the minimum and maximum values in a column; **count**, which counts a column; and **scalar**, which produces a scalar value from a list by taking the first row's first column. Here are some examples of assertions involving these operators:

```
sum { select fees from visit_list } must equal total_fee
    error: "the fee column does not balance";

count { select * from visit_list } must be greater than 0
    error: "this person must have a visit recorded";

scalar { select reason from visit_list where element# is 1 }
        must be "Exam"
    error: "the first visit is always the examination";
```

There are a number of comparison operators that apply directly to list expressions. These fall into two types. The first is composed of the unary operators, which test a list to see if it meets certain criteria:

```
{ select part# from inventory } ascending
    error: "the parts must be in ascending order";

{ select part# from inventory } must be unique
    error: "the part numbers be unique";

{ select connector from con_list }
        series_of (1, max{select connector from con_list}, 1)
    error:
        "the connectors must begin with #1 and be in strict"
        "ascending order, with none missing";
```

The last example above asserts that the list's column must be in ascending order, and that its values start at "1", end at the maximum value of the column, and increment by "1".

The second type of list comparison consists of the binary operators that test for set containment and equality:

```
{select part# from orders}
    must be in
{select part# from inventory}
    error: "an ordered part has no inventory entry";


{select part# from orders}
    must equal
{select part# from inventory}
    error: "all part numbers must be in both lists";

if {select part# from inventory where in_stock is 0}
        contains {<341>}
    then scalar {select due from inventory where part# is 341} > 0
        error: "part 341 must be in stock or on order";
```

Other Features

There are several other list handling features not outlined above. Lists can be built and then saved in user-defined temporary lists. They can then be referenced in subsequent integrity constraints, and deleted when no longer needed. Also, the list constant is considerably more general than described above: each individual value can be any arbitrary scalar-valued expression. This means that users can build lists from every conceivable combination of objects derived from the form.

Conclusions

SEDL has been used in many different applications since 1980, some with only a few forms and only minimal integrity requirements and others with complex integrity constraints on a hundred or more forms. The expressive power of the language has been more than adequate.

The development of SEDL took approximately two man-years, and demonstrated the relative ease of compiler development on the UNIX[1] operating system with its modern compiler writing tools.

The benefits have been substantial. An exploratory study indicated that "coding" productivity increased by at least four to one when compared with PL/I error check modules. End-users have taken to using SEDL, and this significantly boosts productivity over the entire software life cycle. This success led us to implement other special purpose languages in the ODIN system, including languages for report generation, cross-form integrity constraint specification, and data evolution.

There are several enhancements that are desireable. An important one is to allow a value to be checked against a set of permissable values in a database. Another modification is to provide "pseudo-fields" like the current date or the user's identification, which can be used in constraints and assignments.

We have also found that a double implication operator would be helpful:

```
x = "a" iff y = "b"
    error: "...";
```

Finally, list specifiers could be extended to have the full generality of SQL's **select-from-where** construct, but we have found no practical need for this.

Acknowledgements

John Menard made many helpful suggestions in the early language design; Abe Ellenberg wrote the SEDL domain compiler and assembler; and Dennis Meyer wrote the run-time interpreter.

References

[Cham76] Chamberlain, D. D., et al. "SEQUEL 2: a unified approach to data definition, manipulation, and control," IBM Journal of Research and Development, 20, 6 (November, 1976), 560-575.

[Codd70] Codd, E. F. "A relational model for large shared data banks," Communications of the ACM, 13, 6 (June, 1970), 377-387.

[Ditz78] Ditzel, D. R. "Pattern matching for high level languages," SIGPLAN Notices, 13, 1 (May, 1978), 46-55.

[Hamm76] Hammer, M. M. and McLeod, D. J. "A framework for data base semantic integrity," Proceedings 2nd International Conference on Software Engineering,

San Francisco, California, 13-15 October 1976, 498-504.

[McLe76] McLeod, D. J. "High level domain definition in a relational data base system," Proceedings 1976 ACM SIGMOD-SIGPLAN Conference on Data, Salt Lake City, Utah, 22-24 March 1976.

[Pren78] Prenner, C. J. and Rowe, L. A. "Programming languages for relational data base systems," Proceedings 1978 National Computer Conference, Anaheim, California, 5-8 June 1978, 849-855.

[Ston75] Stonebraker, M. "Implementation of integrity constraints and views by query modification," Proceedings 1975 ACM-SIGMOD Workshop on the Management of Data, San Jose, California, May 1975, 65-78.

OFFICETALK-D: AN EXPERIMENTAL OFFICE INFORMATION SYSTEM

BY Clarence A. Ellis AND Marc Bernal

ABSTRACT

WHAT:
Herein is described an experimental Office Information System designed and built at the Xerox Palo Alto Research Center to allow multi-computer experiments in distribution and sharing of control within an office environment.

WHY:
Current office systems go a long way toward aiding the individual user to transact his/her personal office work. Future office information systems need to aid tightly coupled communities of users. Thus, this investigation into communal computing systems is motivated.

HOW:
Using the Alto/Dorado machines, and the Cedar database and programming environment, we have devised a system which allows the flexible manipulation of electronic forms on the display screen of users and helps to coordinate and control the flow of forms between user workstations. Novel facilities implemented in the system include distributed schedulers, dispatchers, office observer workstations, alerters, a data dictionary synthesizer, change agents, and on-line office modeling, simulation and design facilities.

KEY WORDS AND PHRASES

Office information systems, Distributed systems, Office modeling, Office simulation, Database design, Communal computing systems, Graphical user interfaces, Electronic forms, Data dictionary generation.

CR CATEGORIES

G.4, G.3.3, G.1.2, H.3.2, H.6.3, H.2.1, C.4.7, B.2.4, B.2.5, C.1.3

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0131 $00.75

1. INTRODUCTION

This paper describes a prototype Office Information System designed and implemented at the Xerox Palo Alto Research Center to investigate alternatives for distributed control and data manipulation within a communal computing environment. The system consists of 3 components. (1) a user interface which uses display screens to present an electronic desk-top to multiple users, (2) a database sub-system where specifically structured office information is stored, and (3) a modeling sub-system which acts as an office transaction observer, synthesizer, and change agent. Since each sub-system has a different *raison d'etre*, they present different models of the office to the user. These components are integrated by a synthesis program which builds or alters the database from specifications presented via the modelling language; and by a dispatcher program which notifies users of available work by periodically checking and updating the database status. After introducing the background goals and application areas of the system, this paper will devote a section to each of the above topics.

2. SYSTEM OVERVIEW

Within both industry and academia, there is an interest in and need for office information systems [THURB80], [WOHL82] (abbreviated OIS herein). These systems are intended to aid the office worker to manage information by helping to perform tasks such as text editing, forms manipulation, filing, copying, analyzing, and transmitting information. A number of systems of this nature have been developed; some of the well publicized ones include DEJON80, HEWIT80, LYCKL81, SAND80a, SAND80b, TSICH81, WANG80, XEROX81, ZLOOF80. An earlier system built at the Xerox Palo Alto Research Center (abbreviated PARC herein) was OfficeTalk-Zero. It fulfilled many of the above stated requirements and was reported in ELLIS80.

One possible next step after word processors and personal computing environments [FIKES81], [BARBE82] is the notion of an integrated office system which aids many users to share and communicate office data and control information. OfficeTalk-Zero, for example, provided an excellent individual user interface, but did very little to decrease the number of forms within an office or to aid a manager in understanding and tracking workflows within his office. The experimental prototype system described within this paper provides these global aids, and was

implemented to test the feasibility of an OIS in which a large amount of knowledge and control is imbedded within the distributed system. A central element of our OIS is an entity-relation database. Application independent entities (such as the task entity and the activity entity) and relations (such as the precedence relation and the status relation) reside in this database. Also application dependent data resides here too. System organization and concurrency control are implemented to allow multiple users to simultaneously access the database information from different computers. Besides users representing workers in the office, the database is also constantly accessed by observers, dispatchers and other system modules. This database, the MESA programming language and environment, the basic user interface package, and the networking subsystem were all available and heavily used as OIS under-pinnings. Without these excellent software facilities, and without the Alto/Dorado computers [THACK80] and Ethernet hardware [METCA76], it would have been infeasible to carry out this experiment. Ideas which appear particularly useful within an office environment are included in this implementation such as triggers and alerters [MORGA77], office observers, schedulers, dispatchers, a data dictionary synthesizer, change agents, and office modeling, simulation, and design facilities [RICCI81]. We are performing experiments to determine the viability of coordinating and controlling this large amount of information in both centralized and distributed fashion. Results indicate that a distributed scheduler/dispatcher is feasible and helps to decrease bottlenecks due to a centralized database facility.

3. A USER VIEW

A user of this office information system sees a desktop of electronic forms on his* display screen [*since English lacks convenient neutral pronouns, we use he, his, him in this paper with the understanding that the user can be male or female]. This user sees information concerning activities available to be performed by him. He also sees forms representing each of the transactions in progress at his work station. He has available all of the information within the database for browsing (providing database security allows him access). The user is aided by convenient facilities to cause electronic forms to move around the desktop, to grow, to shrink, to scroll and to learn to fill themselves.

The D in OfficeTalk-D refers to the emphasis of this experiment on distributed control and cooperation within a community of users. We see this as a next step beyond the provision of a personal user interface and believe it will have significant impact within offices. A primary new contribution of this system is that it provides users, office managers, and administrators with *communal* office aids which will be described in further detail in later sections of this paper.

The user system is composed of three layers (detailed later) to allow high level, user friendly communication with the database. This organization allows any office user (typically unfamiliar with database systems interactions) to perform his tasks in a manner similar to the way that he always did. A language for office procedure description which can be used easily and flexibly by office workers who are not computer specialists needs to be available. Within our system, units of work performed by a single user (activities) can be specified by "scripts" (see section 6.2). These scripts can be interactive or completely automatic. Also since one person at one work station can play multiple roles within an office (e.g. I may be a researcher, supervisor, and employee), techniques are needed to present various tasks in progress with respect to the various roles in a clear and structured fashion. Although the user interface was not the primary research thrust of this project, it is an extremely important element. If users find this element unsatisfactory, it hinders proper evaluation of other aspects of the system because of a priori negative predisposition. Within our OfficeTalk-D project, we used an interface similar to one reported in previous OfficeTalk experiments [MOTT80].

4. THE DATABASE

4.1 CedarDB

A primary component of OfficeTalk-D is its database management sub-system. OfficeTalk-D was implemented upon the Cedar database management system [BROWN81], which presents an entity-relation data model to its clients. Novel aspects of the Cedar DBMS include its clean separation between transaction management and data management, its distribution of computation and data, and its interface based upon dynamic rather than static storage allocation. This database system was developed at Xerox PARC as part of the CEDAR Experimental Programming Environment Project [MORRI80]; it offers an interface accessible by Mesa Programs. The basic element of this interface is the *Tuple*. This notion is very close to the one used in the relational model. A Tuple is a set of *fields* each of which can contain a value. Each Tuple belongs to a *TupleSet* which is characterized by a number of *Attributes* and each Attribute is defined to have a *Type* (Integer, Cardinal, String, etc). An interesting Attribute Type is the *TupleRef*. It is a reference (or pointer) to a Tuple of any TupleSet. The presence of these TupleRefs makes the system non-relational. This interface provides four basic primitives: creation, destruction, retrieval and storage of Tuples. The database schema (represented by the TupleSets and their Attributes) is described within the database itself in the *TupleSetSystem* or dictionary. This is elegantly implemented as a set of Tuples, one for each existing TupleSet, which are pointed to by all Tuples in their respective TupleSets. This interface also implements the specification of *indexes* which allow the client to define orderings on the sequences of Tuples.

4.2 OfficeTalk-D TupleSets and Organization

Within the database resides system information, user information, application information and status of tasks in progress. We define an office application to consist of units of work, called *activities*, each of which can be performed at a single work station. For example, one can define the pay calculation activity or the check approval activity. Orthogonal to the notion of activity, we also define *tasks* which denote individual transactions which may need to progress through many activities in order to be processed. We next define a *precedence relation* among the activities which specifies that certain activities must be completed before other activities can begin. For example, the processing of the time card for a particular employer may involve 3 activities. The first activity is

filling out the time card; the second activity done at the supervisor station might be approving the time card; and the third activity performed at yet another work station might be calculation of employee pay for time worked. Thus each time card represents a task and the three activities are performed in a strict order by many tasks. The current status of all executing tasks with respect to the various activities is captured within the database via the *status relation*; this relation is highly volatile because as each activity related to a particular task is completed, the status of the task must change. Notice that it is possible to automatically detect which activities can be initiated for a task. If activity A is completed, then the precedence information indicates which activities can be started upon completion of activity A. A scheduler within our system notifies the correct persons when the above event occurs and the task is ready for those persons to perform. This notion of who does what is made precise and alterable within the database as follows. We define an entity tuple called *actor*, this contains the names of all people or automatic servers within the office who are responsible for doing activities. We also define an entity tuple called *role*; this set contains generic names for positions within the office that people must fill, such as cashier, supervisor, or bookkeeper. By defining a relation called *player*, we can map one or more actors to roles. Furthermore, if actor John Jones is absent today, then we can map Sue Smith, who is present, into the role previously performed by John Johns. Roles are associated with activities via a relation called the *performer* relation. Thus, we have a flexible indirect scheme for determining which actors can perform a particular activity. This allows a notification system to notify all concerned actors that a task is available for them. Any one of these actors may be the first to select the task to be done; the task would then become unavailable to other actors playing the same role. This indirect scheme also allowed us to experiment with automatic schedulers which endeavor to determine which of the actors playing a role is most appropriate to perform the particular task. Then the scheduler automatically assigns the task to that actor. Note that appropriateness of actor for task can be a complex decision, depending upon factors like work loads, initiator preference, authority, relative work speeds, and previous task-actor relations.

4.3 Dispatcher

When a user signs onto a machine in this office environment to do his days work, he authenticates himself as a valid actor within the office system. The system can then associate him with those roles that he is capable and authorized to perform. With this structure, a work station procedure obtains work for its actor independent of the rest of the system by tracing from the actor entity through the player relationship to the roles appropriate to this actor. From the roles, the procedure traces to activities performed by those roles using the performer relation. If any of those activities have tasks within the status relation which are waiting, then the procedure will indicate on the users' screen that additional work is available. This user and all other users who likewise map via their role to performers of the waiting activity, all have the option of selecting this task if they are willing and ready to begin performing their activity upon the task. Similar techniques are employed to implement task initiation and termination and to implementation prioritized alerting facilities. Another dispatching scheme with which we are experimenting is one in which the dispatching function is distributed so that whenever an activity is completed, the performer of that activity must check for all "next" activities and schedule them. This decreases the load upon the database since dispatcher modules periodically scanning the database are no longer necessary.

5 USER INTERFACE DEVELOPMENT ENVIRONMENT

Work on the Officetalk-D user interface was greatly expedited throught the use of a display management system called *Tajo*. Tajo controls a set of simultaneously executing utility programs called *tools*. The user employs tools to manipulate windows, menus, and forms. Windows are rectangular regions of the display screen in which text and graphics can be shown. A window usually has a *name frame* which is a rectangular strip at the top of the window with the window's name and other identifying information shown in white. A window may also have subwindows inside of it. Each tool usually has its own window. Tajo assumes that besides a network of display based computers and keyboards, there is also a *mouse* pointing device. The mouse allows the user to direct the attention of the system to a particular area of the screen. As the mouse is moved along the user's desk, a *cursor* moves along the display screen. A cursor is an arrow (or other small icon) which tracks the movement of the mouse and is used for pointing to items on the screen to select them. By causing the cursor to move within a window or form, the user can select that item and perform actions on it. Typically these are performed by pushing one of the three buttons on the mouse. If the middle button of the mouse is pushed, for example, a list of selections, called a *menu* appears on the screen under the cursor. The user can then select any one of the items in the list by moving the cursor over the desired menu item and releasing the mouse button. Selection of different items in the menu causes different actions to occur.

All of the tool display and manipulation facilities are uniform and imbedded within tajo. Thus the user is free to concentrate to a large extent on the development tasks at hand. For example, source files can be checked out, edited, modified, compiled, debugged, re-edited, and recompiled with minimal concern for the mechanics of the switching. Each tool can be rapidly taken up and put down by the user at any time. When not currently in use, each tool retains its context and logical state and thus can be resumed at the point where it was interrupted.

The basic facilities of Tajo allow the user to enter Tools into the system, activate and deactivate them, and discard them. When a Tool is active, it creates a window on the display through which it presents information to the user. The user communicates with a Tool through its window by choosing commands or values from menus, invoking data or command items in forms, and typing or editing text. The location, size, and arrangement of windows on the display is controlled by the user. He is allowed to enlarge, shrink or move the windows around by pointing the cursor to the appropriate parts of the window frame. If the new window position overlaps another window already on the screen, then the new window partially covers the previously visible one(s). Thus the windows

behave as overlapping sheets of paper.

A set of procedures is available within Tajo which allows the user to rapidly create user interfaces for the tools to be manipulated. These procedures allow the specification of the size, the place and the state (active or inactive) of a window. They also provide the possibility to define the contents and the frame of this window. Thus it is possible to create different fields in a window, to specify their place, size, nature (such as command, number, comments, ...), etc. Other procedures allow the creation and manipulation of menus.

6 USER INTERFACE DESCRIPTION

6.1 Level 1: MLIF and DLIF.

These two interfaces enable the user to manipulate the CedarDB primitives in an interactive way. DLIF is an office database definition facility with similar goals to a Data Definition Language in a DBMS, while MLIF is a browser based upon the utilization of electronic forms.

DLIF allows the creation of TupleSets and the definition of each Attribute type. This program is limited to the creation of simple TupleSets because CedarDB does not permit any dictionary modification. Information manipulation (Tuple creation, modification, retrieval) is done by using MLIF, which displays on the screen a form in which each field corresponds to a TupleSet Attribute. An interesting feature of DLIF is the possibility of defining the "looks" of this form during the creation of a TupleSet. This is done by associating a precise position on the form with each Attribute. Note that each form is completely moveable, growable and shrinkable on the screen since we are building upon TAJO. In a similar way, the user may specify the visibility of each Attribute at a specific WorkStation allowing a degree of information security.

Starting MLIF allows the opening of a transaction within the database specified by a parameter (see figure 1). After this operation, the set of TupleSet names that composes this database is displayed on the screen and a set of commands is made available to the user. These commands are activated by using the mouse and each of them takes the current selection as a parameter (see figure 2). The system allows simultaneous manipulation of different electronic forms. Figure 2 shows for instance how to create a Tuple in one TupleSet and to search, in another TupleSet, for the Tuples corresponding to a specific filter defined by the user.

6.2 Level 2: The Interpreter.

Even given the fact that MLIF is particularly convenient to use, we found that an even greater level of abstraction was possible and strongly desirable in order to be useful to non-specialists within office applications. As an example, when an employee decides to fill out his time card, the system imposes upon him a certain number of operations to complete before he can see the time card form displayed on the screen. Operations such as specifying the database name, parameters, and opening the database are required. These operations are not of a classic type in an office and the eventual utilization of MLIF in such an environment would require, for the user, a previous formation to database notions.

One solution to this problem is to build a specific program for each application (fill out time card, check time cards available, etc). An alternative solution to this very heavy method (which is however used in classic DBMSs) is implemented within our interpreter. This is a general solution which allows the user to specify a sequence of selection, editing, and execution steps at his terminal, and have these automatically performed when this activity is interpreted.

The Interpreter is applicable to any program built within Tajo without modifying anything in that program. It simply simulates the actions a user can apply through Tajo's user interface. This is done by reading and interpreting a file called a *script*. This script is a set of procedure calls which perform specific actions on a Tajo program (menus, commands, text selections, window manipulations, etc). The interpreter understands a set of procedures. For example, the *Pause* procedure gives the Tajo program control back to the user, allowing the creation of semi-automatic scripts. Thus the interpreter simplifies the conception of any specific application.

To fill out his time card, an actor may now choose the corresponding script and cause it to be executed by the Interpreter. Thus he will see the form appearing on the screen without taking care of specific DBMS problems.

Such a script will look like the following:

```
ActivateWindow[ --- ];
ModifyItem[ databaseName, --- ];
InvokeCommand[ Connect ];
SetSelection[ Time card ];
InvokeCommand[ Create ];
Pause[ ];
InvokeCommand[ Close ];
```

The creation of such scripts can be done either by using text editor facilities or by using a program which is an extension to MLIF and provides the user the possibility to create scripts by simulation of an application. This program writes out in a file, in terms understandable by the Interpreter, the operation a user had just done. With this program, the activation of the command "Connect!" will have for effect to open a transaction with a database as MLIF could do, and furthermore will write in a predefined script: "InvokeCommand[Connect];".

6.3 Level 3: The Hypervisor

This is the most external layer of the user interface. Its task is to retrieve from the OfficeTalk-D tupleSets (section 4.2) the set of activities ready to be executed and to display, on the actor screen, the name of each of them. The user may select any of the displayed activities by using the mouse pointing device. At this time the associated script is activated and the user may interact with the system via one or more forms inherent to this activity.

Authorized actors may use a facility provided by the system which allows them not to follow the default sequence of activities as it is described in the forms flow graph (see section 7). This is called the "exception facility" and allows, for instance, a sales manager to send a request for goods directly to the shipping and billing department, shortcutting the credit check step to expedite

service on a rush order for a trusted customer. This facility helps provide interface flexibility which is absolutely necessary for efficient and sensitive office operation.

6.4 Interface Conclusions

Research in the area of user interfaces for office information systems tends toward the design of systems where the use of powerful computer configurations is made transparent to the user. This illusion is done in Officetalk-D by a deep structure interface. This interface is certainly not complete; MLIF is still far from an ideal Data Manipulation Language and the artificial intelligence carried out by the scripts is not developed enough to allow complex procedures to be understood by the interpreter. However, our goal was less to implement a complete graphical DBMS interface than to develope an interface sufficient to support experimentation on our distributed systems ideas (and, we must admit, to try out and illustrate some interesting user interface features).

7. THE MODELLING SUB-SYSTEM

In this section, we describe the *information control net* model and how this model is used to provide a global graphical picture of all the activities and tasks currently in progress within an OfficeTalk-D system. Furthermore by combining facilities of the *editor* (section 7.1) and this *observer* (section 7.3), one can build a dynamic *change agent* for editing active office procedures.

7.1 Information Control Nets

Within an information control net model (which we abbreviate ICN), an office is viewed as a set of inter-related office procedures, each of which can be described by a set A of activities, a set R of repositories, and a set of mapping among A and R [ELLIS79]. The activities, denoted graphically by circles, indicate units of work to be done. The repositories, denoted graphically by polygons, indicate some containers of information within the office such as forms, files, or folders. The first function describes precedence ordering among activities, and the other function describes the "used by" relation between repositories and activities. The functions are represented by arcs connecting circles and polygons, further refinements of the model are available for hierarchical description of nested levels of detail, for capturing exception handling within the office, and for time related processing. The small dot and small hollow dot are used respectively to denote forking/joining for parallel activities and choice/decision activities. For example, in the time card processing procedure partially modelled by figure 3, activity 3 (the third circle from the top) is a decision (OR) activity of approval, and activity 4 is a forking (AND) activity to initiate both activities 5 (logging hours) and 6 (issuing pay) which can proceed in parallel. An arc, such as the one from activity 1 to activity 2, means that activity 2 (checking of an employee's time card) can only begin after activity 1 (filling out of the time card) is complete. The diagram shows us that if the approval is successful, then logging and pay slip issuing can both proceed. Further activities after 7 can commence only after both of the above activities are completed. Thus 7 is a synchronizing (AND-JOIN) activity. Although not present in this example, OR-JOIN activities are also available. Squares in figure 3 represent data repositories (forms, folders, and files). Each activity circle has arcs into it from repositories that it draws information from, and arcs from the circle to each of the repositories that it writes information onto. In our example, there is a repository for the time card form, for the ledger book, and for the cash dispursement folder. For example, figure 3 tells us that the activity 2 of checking the time card requires reading the time card repositorybecause there is a dashed arc from that repository.

The ICN model has been derived in the course of office observation and study. The model is derived from industrial engineering models, computer science models, decision science models and from special needs of the office application. The primary objectives are to provide simplicity of description while retaining mathematical tractability. Concerning these goals, it has been possible to perform analyses of (and transformations on) office information flow using the model [ELLIS79]; and it also has been possible to use the model as a vehicle for capturing initial office information flows by drawing ICN graphs on our personal computer displays [RICCI81]. We do this in conjunction with office workers and managers, and have observed that this structured view of activity within the office is useful and understandable to the people involved in these activities. In fact, these people play an important role as correctors and editors of our frequently incorrect and incomplete initial models of their office. An extremely useful feature of our ICN system is its graphical editing capability. Thus a user can point at an activity circle or arc on the screen and delete it, move it or edit it in various ways.

7.2 Simulation

The next phase in creating and designing a set of office procedures from an ICN is the simulation phase. The non-static part of an office is the set of tasks or work flow which is constantly coming and going. These tasks are represented on the screen by tokens which are created by the simulation system, the timing and flow of simulation tokens are determined by user input parameters, and the ICN structure respectively. The simulation thus consists of running many tokens through the specified office procedures. Typically this is done in a faster than a real time mode. Statistics are automatically gathered concerning the work load on the various simulated participants and concerning waiting times and queue lengths for tasks. This type of simulation gives top level information concerning measures such as throughput, turn-around time, and those parts of an office which may be too loaded with work to run smoothly. Information gained from a simulation can be immediately used to edit the model and then run further simulations and edits. At some stage, more detailed simulation and testing may be desired; this is accomplished by distributing the simulation across many machines connected by the Ethernet. When this is done, each machine acting on behalf of a particular set of people may execute a procedure which carries out the semantics of the simulated activity (e.g. filling out a form). At this stage, it is possible to replace a simulation at a machine by a human interacting with the system via OfficeTalk-D. This implements a nicely *controlled environment* for observing the reactions, responses, and problems encountered by an individual under carefully regulated workload.

7.3 Observer

A novel aspect of our system is the capability of managers or office workers to see a dynamic graphical view of work progress within their office. The provision for one or more observers within OfficeTalk-D is a natural extension of the simulation facility previously described. Instead of tokens representing simulated tasks they now reflect actual tasks or subtasks, and they can be selected to display their history and statistics. Instead of having token generation and movement artificially controlled by a simulator, we allow the system dispatcher to trigger the introduction and movement of tokens whenever it is detected that work within the office is available or has been completed. When these changes are noticed by the observer, it moves the token corresponding to that task from the old activity to the new activity. In the general case, it may be necessary for many "old activities" to be completed, before the new activity or activities can commence. This phenomenon, and the fact that users may choose not to select a waiting task to work on, implies that the observer sometimes will show task tokens queueing on arcs. Several tokens may be associated with one task. Since the observer (and the simulator) must reflect the outcome of decisions and of concurrent activity on tasks, it is necessary to remember the identity (and other characteristics) of tokens.

8. OFFICETALK-D FEATURES

In this section we mention some of the features within our OIS which can make life more congenial and expeditious for an office community. There are many tasks within the office setting which make this newer mode of operation quite valuable; in fact the real pay-off for the "office of the future" will come when integration has reached a high enough level that communal aids for co-operation and control can be applied on a large scale. We can pinpoint some common types of applications in which there is a particularly strong need for tightly coupled computing as is provided by OfficeTalk-D:
(1) Bargaining - This is typified by the customer-salesperson relationship in many offices. To be carried out effectively over long distances it really requires tightly coupled communications.
(2) Problem Solving - Distributed facilities for communal problem solving are needed. Some of the most frustating aspects of office work are concerned with exception handling. This is a classic case of problem solving; it is apparent that facilities for global sharing and synchronization can ease this task.
(3) Multitasking - Frequently many people are performing the same procedure on different data or for different customers. To do this effectively, it must be possible to simultaneously access the same control information and to synchronize. Thus sharing and control facilities and aids again are valuable.

8.1. Concurrency Control

The file system upon which our OIS is built provides page level access, caching on the user machine, and access modes of [read, write, extend, create]. Within this file system, the original locking mechanisms were only provided at the file level. One aspect of our research objective was to help determine which concurrency control mechanisms are appropriate for this type of application. The simplest mechanism that we tried was file level locking, but the resultant long delays experienced by users was intolerable. Our belief is that a real need exists for multiple users to be able to simultaneously access information (perhaps the same information), but conflicts causing inconsistency are actually rare. This implies that unstructured simultaneous access with NO concurrency control works MOST of the time and is very efficient. We tried this scheme and inconsistencies occurred at unacceptably frequent intervals. One reason for this is the caching mechanism which gives the effect of very long transactions (until the cache is eventually written back before a *close-database*). If a second user reads data into his cache e.g. via *open-database*, while a first user has his database open, then the first of these to *close-database* will lose his update information when the second *close-database* overwrites it. The delay due to locking could be reduced somewhat by causing the database to be formed as multiple files. Unfortunately, the database facility for doing this naturally caused all tuples in the same tupleset to be placed in the same file. We found that the logical division of data frequently does not correspond to the physical division requirements. Several tuples, such as the status tupleset were accessed by many users and so the bottlenecks were still present. Notice however that although many users access the status tupleset simultaneously, most only access their own tuple. This suggests the use of fine-grained locks. We implemented a new mode on the file server called Writeshare mode, and then used software to implement fine-grained locks. The improvement in performance was significant and since the lock information was in Write mode, we were assured of database consistency.

8.2. Alerters

OfficeTalk-D includes an alerting facility which allows the office worker to specify exceptional conditions that are of interest. Whenever an exceptional condition that has been specified arises, the system will invoke a user specified procedure associated with that exception. Alerting can take place upon database items that assume exceptional values, or upon time constraints that have been exceeded. Examples within a time card processing environment include alerting the supervisor if it is Friday afternoon and any of his workers have not submitted a time card; and alerting the accountant if any worker fills in the HOURS field of his time card with a number greater than 40 hours.

8.3 Data Dictionary Synthesizer

Given an ICN, it is possible to build the database of OfficeTalk-D basic system tuplesets (data dictionary) automatically. OfficeTalk-D contains a module to translate from the ICN model to the entity relation database model which is used to aid in the movement from design and experimentation on a simulated system to an executing environment. Some of the mappings done by this database synthesizer are obvious. Each activity circle in the ICN causes creation of a tuple in the activity tupleset. An arc from an activity A to an activity B causes creation of the tuple [A,B] in the precedence relation tupleset. "Done by" labels in the ICN each correspond to a tuple in the actors tupleset, and the preformer relation between actors and activities is similarly established by noting the "done by" label for each activity circle in the ICN. Finally, repositories are

the ICN items which map to forms and application dependent data within the database to be manipulated by DLIF and MLIF (section 6).

9. CONCLUSIONS AND REMARKS

9.1 Research Directions

We are currently exercising and measuring OfficeTalk-D and hope to report results in a future paper. This distributed office information system has potential advantages of reliability, homogeniety, changeability and other features accredited to distributed systems. Some fruitful areas of research are concerned with (1) user interface engineering, and (2) development of techniques for dynamic change. We feel that a natural extension of the observer is to graduate him to status of "change agent". Currently we can define a region of change for any sequence of edits to an ICN and flush this region of transactions (by waiting for all tokens to leave, and not allowing any new tokens in). Then we treat the edits as an atomic transaction, and perform the changes, and then resume the office system. It is highly desirable, but a tough research problem to do better than this somehow.

A primary disadvantage of the current organization is the heavy reliance upon the database for information and communication. Notice that bottlenecks can occur because all users, dispatchers, and observers are required to periodically examine various tuplesets such as the status relation in the database. An alternative to this architecture is to require that work stations send forms and messages directly to the successor work stations (distributed scheduler) thereby requiring knowledge of the location of others in the system. We are currently investigating another fascinating approach which imbeds default knowledge within forms processes which are the representations of tasks within the office system [ELLIS81]. With this approach, migrating forms processes contain much of the intelligence of the system; processors and repositories then become resources (just like page frames in operating systems) to be scheduled and available for load-sharing. This technique holds promise of allowing maximum distribution of information and control while avoiding constipation within the database access interface.

9.2 Acknowledgements

OfficeTalk-D was designed and implemented by Marc Bernal, Andy Daniels, and Clarence Ellis. This implementation would not have been possible within the 6 months that we worked if it were not for the hard work of many systems designers and implementers at Xerox PARC. We would like to thank many people within the old Systems Sciences Lab and at PARC PLACE for stimulation, discussions, and encouragement (particularly Bert Sutherland and Gary Nutt). We would like to thank many people within the Computer Sciences Lab (particularly Mark Brown and Rick Cattell) and within the Systems Development Division (particularly Ted Wobber and Mark Sapsford) for their software and for their patience and help. We would like to thank the INRIA of France (particularly Najah Naffah) for providing payment and support for Marc Bernal while he was a visiting scientist at PARC. Finally we would like to thank Xerox Corporation for the generous support of this wonderful computing environment.

10 REFERENCES

BARBE80 Barber, J., "Reasoning about Change in Knowledgable Office Systems", Workshop on Research in Office Semantics, Chatham, Massachusetts, June 1980 (also PhD Thesis, MIT-AI Lab).

BARBE82 Barber, J., "Supporting Organizational Problem Solving with a Workstation", Proceedings of the ACM-SIGOA Conference on Office Information Systems, Philadelphia, Pennsylvania, June 1982.

BROWN81 Brown, M. R., Cattell, R. R., and N. Suzuki., "The Cedar Database System" Proceedings of the 1980 ACM Conference on the Management of Data, Ann Arbor, Michigan, May 1981.

DEJON80 DeJong, S. P., "The System for Business Automation(SBA): A Unified Application Development System", Proceedings of the IFIP Congress 80, August 1980.

ELLIS79 Ellis, C. A., "Information Control Nets: A Mathematical Model of Office Information Flow", Proceedings of the ACM Conference on Simulation, Modeling, and Measurement of Computer Systems, August 1979.

ELLIS80 Ellis, C. A., and G. J. Nutt., "Office Information Systems and Computer Science", ACM Computing Surveys, March 1980.

ELLIS81 Ellis, C. A., "An Office Information System Based on Migrating Processes", INRIA International Workshop on Office Information Systems, October 1981.

FIKES81 Fikes, R., "Odyssey: A Knowledge-Based Assistant", Artificial Intelligence Journal, Volume 16, Number 3, July 1981.

HEWIT80 Hewitt, C., Attardi, G., and M. Simi, "Knowledge Imbedding in the Description System Omega", Proceedings of the First Annual National Conference on Artificial Intelligence, August 1980.

LYCKL81 Lycklama, H., "An Integrated Office System Based on UNIX", IEEE Lake Arrowhead Workshop on Office Information Systems, Lake Arrowhead, California, September 1980.

METCA76 Metcalfe, R. M., and D. R. Boggs., "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, Volume 19, Number 7, July 1976.

MORGA77 Morgan, H. L, and O. P. Buneman, "Implementing Alerting Techniques in Database Systems", Proceedings of the AFIPS National Computer Conference, 1977.

MORRI80 Morris, J. B. "The CEDAR Experimental Programming Environment", presented at the ACM National Conference, Nashville, Tenn, 1980.

MOTT80 Mott, T., "Design, Implementation, and Installation of an Office Information System" OFFICE80 Symposium on Office Information Systems, University of Toronto, Canada, September 1980.

NAFFAH81 Naffah, N., "The KAYAK Office Information System Research Project" presented at the SOGESTA summer school on Office Information Systems, Urbino, Italy, September 1981.

NUTT79 Nutt, G. J., and C. A. Ellis., "BackTalk: An Office Environment Simulator", Proceedings of the 1979 ICC, volume 2, June 1979.

NUTT80 Nutt, G. J., and C. A. Ellis., "On the Equivalence of Office Models", Xerox Palo Alto Research Center Report No. SSL-79-8, December 1979.

PURVY82 Purvy, R., Farrell, J., and P. Klose "STAR Records Processing: Data Processing for Non-Computer Professionals", Proceedings of the ACM-SIGOA Conference on Office Information Systems, Philadelphia, Pennsylvania, June 1982.

RICCI81 Ricci, P. A., and G. J. Nutt., "Quinault : An Office Modeling System", IEEE Computer Magazine, Special Issue on Office Automation, June 1981.

SAND80a Sandelin J., Kanerva, P., and J. H. Siegman., "Specification of a Computer-based Text System" SIGOA Newsletter (ACM Special Interest Group on Office Automation), Volume 1, Number 2, May 1980.

SAND80b Sandewall, E. H., et. al., "Provision for Flexibility in the Linkoping Office Information System", Proceedings of the AFIPS National Computer Conference, June 1980.

SIRBU82 Sirbu, M., "Understanding Managerial Group Work", Proceedings of the Third AFIPS Office Automation Conference, April 1982.

THACK80 Thacker, C. P, McCreight, E. M, Lampson, B. W, Sproull, R. F, and D. R. Boggs, "Alto: A Personal Computer", *Computer Structures: Readings and Examples*, Siewiorek, Bell, and Newell, Eds, 1980.

THURB80 Thurber, K. J., (editor) *Tutorial Office Automation Systems*, Computer Society Press, IEEE Computer Society, 1980.

TSICH81 Tsichitzis, D., et. al., "Omega Alpha", University of Toronto, Computer Systems Research Group, Technical Report CSRG-127, March 1981.

WANG80 "The Wang Corporation Office Information System" in The Seybold Report, Media, Pennsylvania; Seybold Report on Word Processing, v.2, no.12, 1980.

WOHL82 Wohl, A., "Office Automation Systems: Integration Issues", Proceedings of the Third AFIPS Office Automation Conference, April 1982.

XEROX81 "Xerox's STAR" in The Seybold Report, Media, Pennsylvania; Seybold Publications, v.10, no.16, 1981.

ZLOOF80 Zloof, M. M., "A Language for Office and Business Automation", Proceedings of the First AFIPS Office Automation Conference, March 1980.

Figure 1

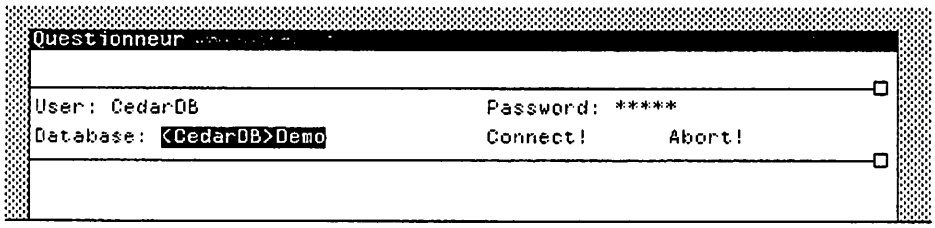


Figure 2

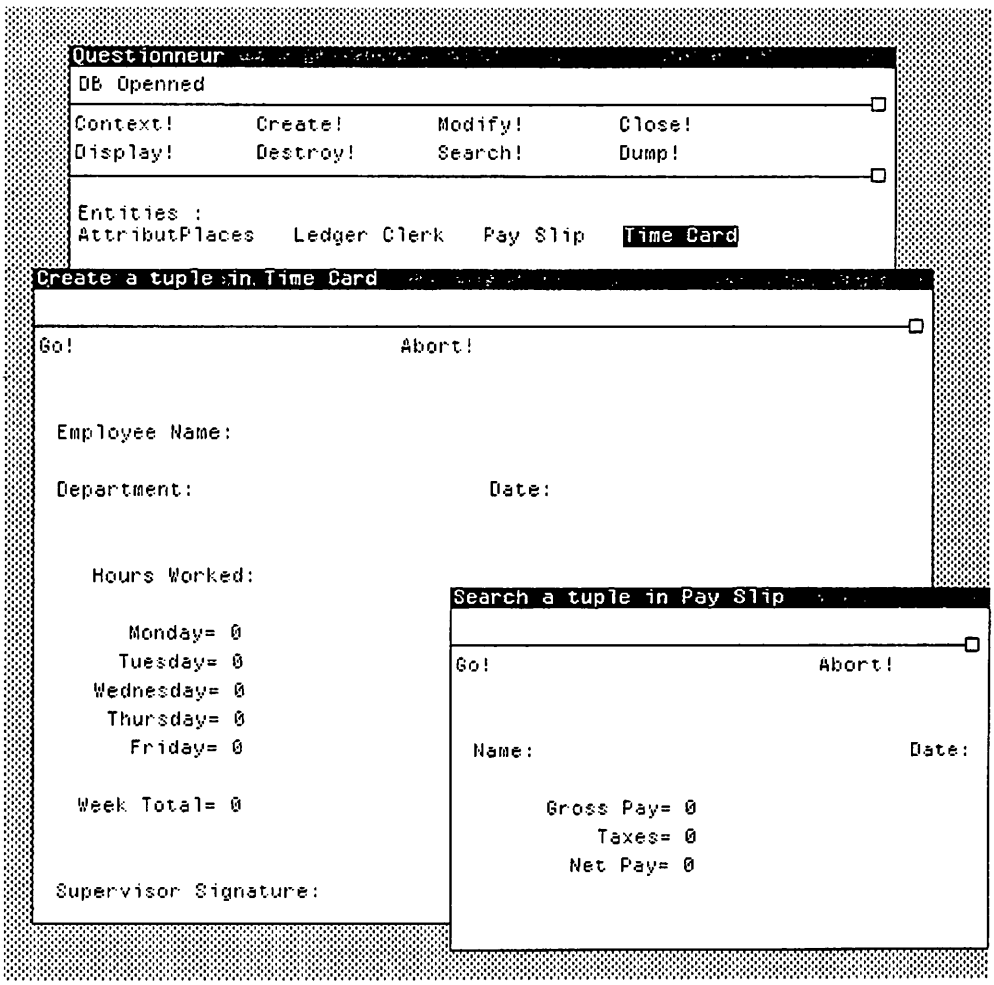
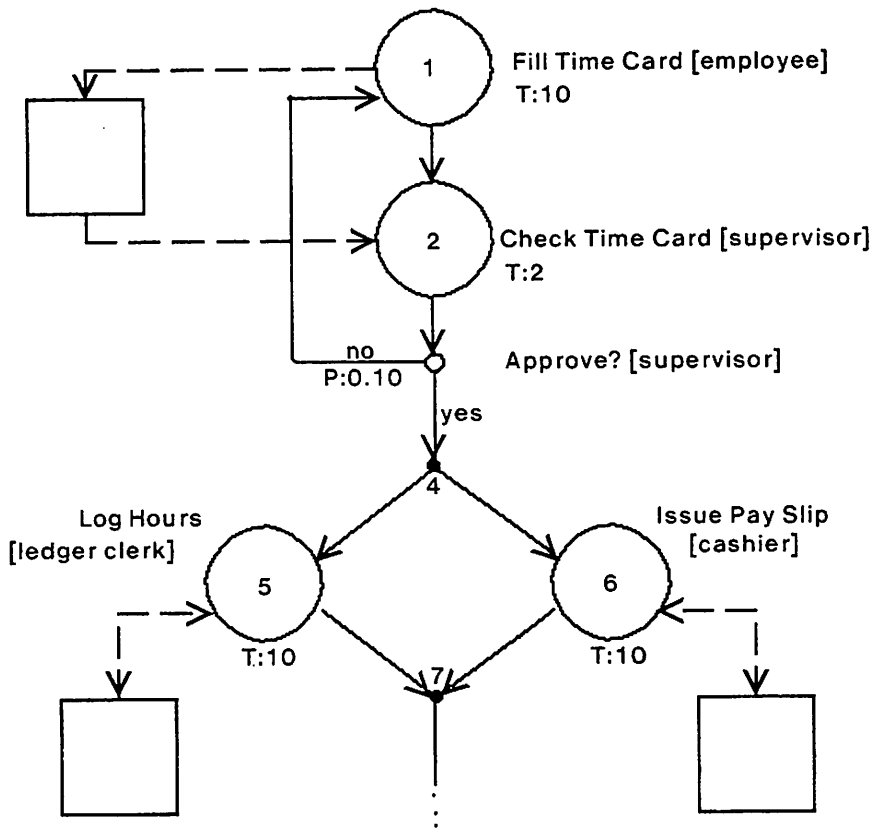


Figure 3

INFORMATION CONTROL NET

(time card processing)



T: denotes time parameters (for simulation)

P: denotes probabilities

A Modelling Tool for Office Information Systems

J. Lebensold, T. Radhakrishnan and W.M. Jaworski

Computer Science Department

Concordia University

Montreal

ABSTRACT

Application of ABL, Alternative Based Language, to modelling office information systems is described. This language is powerful for the description of parallelisms and is amenable to stepwise refinement. After studying the different OIS modelling tools, we propose eight requirements of a good modelling tool and examine how well ABL fulfills these requirements. Some capabilities of the ABL environment and user interface are demonstrated. We present the ABL description of an order processing office.

1.Ø The Office Environment

In an office, people receive, deal with, classify, remember, find and send information (SCHE81). The transmission of information is done orally as well as on paper. Uhlig et al. point out that besides office workers processing information, they also generate new information (UHLI79).

Ellis defines an office "...as a set of related procedures. Each procedure consists of a set of activites connected by temporal orderings called precedence constraints. In order for an activity to be accomplished it needs resources - examples include files, calculators, people, pencils, telephones" (ELLI79). According to Tsichritzis an office procedure "consists of three parts: a condition, an action and a notification part. The condition part specifies the terms under which the procedure is initiated. The action part specifies the operation that the procedure performs when it is initiated. Finally, the notification part specifies the other objects which are notified when the procedure completes" (TSIC8Ø). Of note is SSA's (Structured Systems Analysis) definition of a business "as a logical set of functions which exists to provide a product or service..." (MEND8Ø). Uhlig et al. maintain that "... viewing the office as a communication system enables us to describe the process in quantitative terms" (UHLI79).

Faced with these varying considerations of what we mean by "office", we prefer the following interpretation for our discussion: an office is that part of an organization into which comes information, from which comes information, and in which information is generated or transformed in such a way that it can be used outside the office to produce products, services and money. The information that enters an office may come from many different sources, some of which might be considered non-traditional from the point of view of computer system input. There can be both structured and non-structured input data entering the office through different channels such as speech, vision, gestures and mannerisms.

Given that interpretation, we can characterize offices by the kinds of information that enter them, the kinds that leave, the kinds that are generated, and the sorts of transformations that take place within them. Also, we can look at the types of tools that are used both in the input/output operations and in the transformative operation.

For example, an order processing office (ELLI79) becomes a place into which order information comes, and is subsequently transformed, using various files, into actual orders, bills and delinquent notices. Of course, we should not forget that "in the end, it is people, not machines, who make an office what it is" (UHLI79).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-075-3/82/006/0141 $00.75

2.Ø Requirements

After examining the strengths and weaknesses of several different OIS modelling systems we have formulated the following eight points as the desirable features of a modelling tool for an office information system:

1) Be simple to learn and use.

Ideally, the modelling tool should be able to be used not only by the OIS professional, but also by the average office worker and manager. This condition necessarily implies a "friendly" user interface. The question of what constitutes such an interface for the office environment modelling is one that requires careful study and analysis, but it is beyond the scope of this paper. However, such an interface will, at the very least, allow decision-makers within the organization to understand the models produced. It is indeed possible that a modelling tool gains wide acceptance mainly on the basis of its user interface quality.

2) Represent reality closely.

One of the major problems with many modelling tools is that once a model has been established, it is not straightforward to realize the system being modelled. It is well-known that one cannot simply scale up a model and hope that it will work as specified. Similar problems can occur when we try to translate a model into reality. A model that could be directly implemented would be the ideal. However, given that it is highly unlikely that we will be able to automate every activity in the office, there should be at least the capability of directly mapping the model to reality. Those parts of the office that are amenable to automation should be able to be executed directly from the model. In this way, if the actual system proves to have flaws, then we would be able to relate them to the model and rectify it. The process of system design then becomes interactive and dynamic. This approach represents an improvement over the usual method which runs as follows: 1) develop the model; 2) translate the model into a set of programs that can be executed; 3) run the programs; 4) if the system does not perform as specified then either the model is at fault or the translation is not faithful to the model, or both. At this point we need to be an expert in both the translation process and model building. By eliminating, or greatly diminishing the translation step, we are able to concentrate on the flaws in the model itself.

3) Be consistent at various levels of the office hierarchy.

Business organizations are traditionally structured hierarchically. The model should be able to reflect that hierarchy. However, at various levels of the organizational tree, there can be functions and procedures that are not hierarchical in nature. Without resorting to different tools, the model should be able to reflect both hierarchical and heterarchical structures.

4) Express both asynchronous and concurrent events.

As Uhlig et al. point out, "Processes occur in parallel, and clustered together, represent the various organizational functions" (UHLI79). The model should be able to represent office acivity adequately, whether it is sequential, parallel, or a combination of both. Individuals themselves carry on activities in a parallel fashion, doing more than one thing at a time. Also, the model should be able to represent the many kinds of interrupts or asynchronous events that occur within an office.

5) Have a sound theoretical base.

In order to facilitate the analysis of whatever systems are being described, the modelling tool should have a sound basis in theory. It would be preferable if this basis could be quantified mathematically. That way, the verification of a model reflecting both existing and proposed systems would be much easier.

6) Represent both "document model" and "processor model".

An office can be looked at in two quite different ways. One is to consider the kind of activities that occur, such as filing, mailing and so on. The other is to examine the kinds of data that travel through the organization, such as reports, documents and forms. The model should be capable of representing an office in either fashion, or, indeed, in both at the same time. In an actual business office, neither the documents nor the processes that deal with the documents are separated from each other. The model should be able to deal with either of the two approaches.

7) Handle incomplete specifications.

One of the major uses of an OIS modelling tool is to plan new offices and upgrade existing ones. It is not unusual for the specifications to be incomplete. Often this incompleteness becomes apparent after part, or in the worst case, all of the implementation has been done. A good modelling tool should help to bring out

inadequacies in the specifications or in the implementation.

8) Allow incremental change of models.

It should be easy to make changes to an existing model as soon as any inadequacies are discovered. Rather than reconstructing the model from the beginning, it would be better to modify it, one step at a time, and verify its performance at that stage before proceeding.

3.0 Three Existing Tools

3.1 SSA - Structured Systems Analysis

SSA was developed as part of an effort by Exxon Corporation to help improve software design and implementation. Mendes describes SSA as a "business modelling and communication technique" that is used by both the systems analyst and the business person (MEND80). SSA concerns itself with both the actual model that is built and the process of building the model.

The SSA model includes a "global model" which is a hierarchy diagram, a "function matrix" which is an NxN array, an "information flow diagram" which is a network diagram, a "detail activity model" which is an annotated hierarchy diagram, a "data structure diagram" which is another annotated hierarchy diagram, and a "glossary of business terms" which is an English language narrative description.

The global model describes how the various functions in the organization are logically related to each other; the function matrix defines the responsibilities associated with each function; the information flow diagram represents how information flows through the organization; the detail activity model describes the "lowest-level functions in the Global Model"; the data structure diagram describes the view that the business person has of the information in the organization; and the glossary defines whatever terminology is relevant to that business.

3.2 ICN - Information Control Nets

An Information Control Net (ICN) is a graphical representation of the "set of related procedures" that makes up an office (ELLI79). It is a flow model that has a rigorous mathematical description. The formal description is presented in detail in ELLI79. This modelling tool is based on Petri Nets (PETE77). The graphical representation is built up using circles to denote activities, squares to show data storage facilities, solid arrows indicating which activity comes before which other activity and dashed arrows to indicate the storing and retrieval of information. Arrows coming from nowhere are points of initiation, and arrows going nowhere are points of termination. Parallelism is indicated through the use of "AND nodes" which are solid circles from which come arrows pointing to the various parallel activities. Finally, there are "decision nodes" which are circles into which come dashed arrows, and from which come solid arrows pointing to the activities between which a choice is to be made. The ICN is modular in that any circle can represent a complex activity (see Figure 1).

3.3 BDL - Business Definition Language

Business Definition Language was designed to be a problem-oriented, readable, and easy-to-modify "data processing" descriptive language (HAMM77). It was developed for communicating not only with computers, but also with people. BDL follows the structure of offices very closely, mirroring the flow of data through an organization.

There are four objects that BDL recognizes: documents, steps, paths and files. A document is the basic data structure, "simply data filled out on a form." Steps correspond to the "organizational units of the system being described", and they show, at the lowest level, how input documents are transformed into output documents. Paths represent the data flow. Documents flow over paths to get from one step to another. Files are "permanent repositories of documents." Since a document only exists between steps, it is necessary to have such a facility for storing information permanently.

Three of the major components that make up BDL are presented in HAMM77. They are the Form Definition Component (FDC), the Document Flow Component (DFC), and the Document Transformation Component (DTC). Rather than there being one language which is used throughout BDL, there is a specific language for each of the separate components. Also, BDL programming requires an interactive terminal with some graphic facilities.

3.4 Evaluation

Table 1 summarizes our evaluation of SSA, ICNs and BDL with respect to the eight requirements stated in Section 2. For want of space only the summary is given here and a detailed analysis can be found in LEBE81.

4.0 ABL - Alternative Based Language

ABL was developed out of a concern for

TABLE 1

AN EVALUATION OF THREE MODELLING TOOLS

(Summary of analysis in LEBE81)

REQUIREMENT LEBE81	SSA MEND80	ICN ELLI79	BDL HAMM77
1-Simplicity	4	2	4
2-Accuracy	5	5	4
3-Consistency	3	1	1
4-Parallelism	2	1	1
5-Theory	1	1	2
6-Document/ processor	1	3	1
7-Incomplete specs	3	2	1
8-Incremental change	2	1	3

KEY: Correspondence between method and requirement:
1 - Excellent, 2 - Very good, 3 - Good, 4 - Fair, 5 - Poor

the need to have reliable software (BELK76, HINT81). There are structural similarities between ABL, decision tables (MCDA7Ø) and guarded commands (DIJK75), although they differ in many respects. In addition to providing the software designer with a powerful programming tool, ABL also allows the system model builder to describe clearly and concisely how any system behaves. This section describes the use of ABL as an OIS modelling tool. In the following pages, syntactic units of ABL are contained within angular brackets. The semantics of ABL will become clear when we present a sample case in Section 5. In the sequel we examine how well ABL fulfills the eight criteria we have proposed.

1) ABL has been used by undergraduate students taking a first computer course for analysis and problem solving. From our experience it is found that students who have no preconceived notions about computer programming have an easier time understanding and using ABL than those who have been exposed to conventional programming languages like FORTRAN, COBOL or Pascal. Work is continuing on the development of a wide range of interfaces including those requiring the use of special graphics terminals. Also, libraries of <System>s, both simple, consisting only of <Action>s, for example, and complex, consisting of <Abstract machine>s and <Flow>s, are being built up.

2) By successively refining an ABL <System>, it is possible to maintain the same structure in a model as in the implementation of that model. <System>s are successively refined until they become either <Description>s or other <System>s. In the latter case, the <System>s that result can themselves be further refined. The <Description> then can either be executed by a machine, or interpreted by a human being.

3) ABL permits the user to partition a <System> into several parts. Both the relationships that exist between the parts, and the parts themselves, can be displayed in various ways, depending on the choice of user interface. Thus it becomes possible to look at an office environment both from a "low-level" point of view and from a "high-level" one. The OIS modeller can focus on any aspect of the office environment.

4) The construction "(<a>,<b>)" in the syntax definition of ABL shows the capabilities of ABL for dealing with concurrency. An <Alternative>, on the other hand, is a sequence of <Action>s. An interrupt is a particular kind of <Action>, and the state of the system at the time it occurs is a <System>. This <System> is a <Data object> which can be stored for recall and activation after the interrupt has been serviced.

5) The formal roots of ABL are to be found in the relational algebra, decision table theory, and formal language theory.

6) Since a <Flow> may be either a <Data flow> or a <Control flow>, or both, ABL gives the OIS designer complete flexibility in terms of being able to represent either the activities in an office, or the data that flows through it.

7) Because ABL permits complete freedom in dealing with the various levels of an organization, incomplete specifications are easily handled. The "ELSE" clause in decision tables is the underlying power of this facility to deal with incomplete specifications. This capability is implemented as a transfer of control to an <Exception step>. We are also extending ABL to include post-conditions that are pre-defined by the designer. These are evaluated after the execution of each <Alternative>, and control is subsequently transferred either to a normal <Step> or to an <Exception step>, depending on the state of the post-conditions.

8) An ABL <System> is easy to modify at specification, design and implementation stages. <Action>s, <Predicate>s, <Data

object>s, <Alternative>s, or <Step>s can be added or deleted at any time during the development of a program.

5.Ø A Case Study

We present the following case study drawn from ELLI79 (p.226). The example referred to, described in Figures 2 through 5, concerns the order processing department of a company. A software tool is available at Concordia University to produce such figures from the input specifications of a problem.

This case study does not bring out all the capabilities of ABL. A broader scope of ABL can be found in both the Appendix and in LEBE81 where examples from MEND8Ø, ELLI79 and HAMM77 are described in ABL.

Figure 2a

The first two lines are the <Narrative> that describe the entire <System>. They are repeated in each of the following displays. This listing shows all the <Step>s. In this case, there are two <Step>s. The numbers between "<" and ">" represent <Next step>s, each one corresponding to one <Alternative>. This <System> is composed of seven <Alternative>s. The <Step> Ø indicates an empty <Next step>; that is, the program terminates.

Figure 2b

The same information is presented as in Figure 2a, with the addition of <Predicate>s, each numbered set of individual <Predicate>s corresponding to an <Alternative>. Note that the two <Predicate>s associated with <Alternative>s 2.1, 2.2, 2.3, and 2.4 are the same. Of note here also is ABL's facility for representing parallelism shown by the two <Alternative>s, 1.1 and 1.2.

Figure 2c

The information presented in Figure 2b is expanded by the addition of the <Description> of each <Alternative>. Furthermore, each of these <Description>s is followed by an arrow pointing to the <Next step> in the <Control flow>. <Alternative>s 1.1 and 1.2 result in <Step> 2 with a corresponding re-evaluation of the <Predicate set>. Each <Alternative> in <Step> 2 results in termination.

Figure 3

In Figure 3 we have added the <Action>s that make up each <Alternative>. Each <Action> and each <Predicate> here is a <Description>. In this case the <Description> is a <Narrative>; however, if the <Description>s were each <Host language code>, that is one or more instructions in a programming language, then this <System> would be executable on a machine. For example, <Alternative> 2.2 is made up of three <Action>s, namely "FILL OUT FORM F1", "FILL OUT FORM F2" and "BILL LATER". Note that <Alternative> 1.3 is essentially an empty <Alternative>. We see here that the two <Alternative>s 1.1 and 1.2 actually consist of different <Action>s; however, they are to be executed in parallel. <Action> 2.3A is a complex <Action>. When it is executed, control is transferred to the corresponding <System>. It is important to note that the other <System> can consist of the same <Abstract machine> as the original <System>.

Figure 4

Until this point, we have been adding information to the <System> presented in Figure 2a. Figure 4 "separates" <Predicate>s from their values (true(Y), false(N) and irrelevant(-)) defining under what circumstances which of the <Alternative>s will be chosen. This figure shows how the <Alternative>s are selected. In this instance, if <Predicate> "CUSTOMER REQUEST ARRIVES" evaluates true(Y) then <Alternative>s 1.1 and 1.2 will be chosen. Note that the <Next step>s are still indicated, as are the individual <Action>s.

Figure 5

Figure 5 is perhaps the most comprehensive presentation of the <Control flow> of this <System>. The description of the <Step> is at the top, followed by the description of the <Alternative>s that make up that <Step> with the corresponding <Next step> listed to the right. In the table that follows, under the heading "CONTROL FLOW", each column that has a "V" in the row labelled "C1" corresponds to one of the <Alternative>s of <Step> 1. The <Action>s making up the <Alternative> are numbered in the order in which they are to be executed and sorted in that order. Both <Action>s and <Predicate>s are listed under the heading "MACHINE". This is the <Abstract machine> for this <System>. The <Next step> is indicated at the bottom of the table. Thus, for example, <Alternative> 2.2, "SHIP NOW, BILL LATER", is chosen when <Predicate> "SEND C.O.D." evaluates false and "BILL LATER" evaluates true. It consists of <Action>s A7, A8 and A6, in that order. Since the <Next step> is Ø, the <System> stops.

Descriptions of the warehouse department example from MEND8Ø and the accounts receivable example from HAMM77

using ABL can be found in LEBE81.

6.0 Summary

We have proposed eight requirements of a modelling tool intended for office information systems and examined how well these requirements are fulfilled by three well known modelling tools published in the literature. ABL, Alternative Based Language, is examined in the light of these requirements. This language is rich in constructs to express parallelism and modularity and permits the user to interact with the model in a variety of ways (KRON82). Work is in progress to provide an environment for direct execution of ABL models of office information systems. Although we consider the eight requirements stated in this paper to be necessary, their sufficiency is yet to be verified.

Acknowledgements

In addition to the constructive comments of the referees, we are grateful to the following people for their assistance and encouragement: Dr. T. Fancott, L. Ficocelli, D. Eddy, and the other members of the "ABL group" in our Computer Science department.

REFERENCES

BELK76 Belkin, G. and Jaworski, W.M. "Towards Logic and Performance Analysis of Unwritten Programs", Conference Proceedings, Canadian Computer Conference, Session '76. Montreal.

DIJK75 Dijkstra, E.W. "Guarded Commands, nondeterminacy and formal derivation of programs", Communications of the ACM, Vol.18, No.8, August 1975.

ELLI79 Ellis, Clarence A. "Information Control Nets: A Mathematical Model of Office Information Flow", 1979 Conference on Simulation, ACM Proc. Conf. Simulation, Modeling and Measurement of Computer Systems, Aug. 1979, 225-240.

ELLI80 Ellis, Clarence A. and Nutt, Gary J. "Office Information Systems and Computer Science", Computing Surveys, Vol.12, No.1, March 1980.

HAMM77 Hammer, M., Howe, W.G., Kruskal, V.J., and Wladawsky, I. "A Very High Level Programming Language for Data Processing Applications", Communications of the ACM, Vol.20, No.11, November 1977.

HINT81 Hinterberger, H. and Jaworski, W.M. "Controlled Program Design by use of the ABL Programming Concept", Angewandte Informatik, 7/81, 302-310.

KRON82 Kronick, Murray N. "A Software Environment for a Mini-computer", Master of Computer Science Thesis, Concordia University. March 1982.

LEBE81 Lebensold, J. and Radhakrishnan, T. "Modelling and Implementation of Office Information Systems: A Comparison of Approaches", Computer Science Department Technical Report, Concordia University, 1981. Available from the authors.

MCDA70 McDaniel, Herman. Applications of Decision Tables, Brandon/Systems Press, Inc., Princeton, 1970.

MEND80 Mendes, Kathleen S. "Structured Systems Analysis: A Technique to Define Business Requirements", Sloan Management Review, Summer 1980.

PETE77 Peterson, James L. "Petri Nets", Computing Surveys, Vol.9, No.3, September 1977.

SCHE81 Scheurer, B., Viarnaud, M.L., Mantoux, G., Berber, D. & Querard, B. "Le buroviseur, poste de travail dans le bureau du futur", Bulletin de Liason de la Recherche en Informatique et Automatique, No.70, 1981, pp.2-8.

TSIC80 Tsichritzis, D. "OFS: An Integrated Form Management System" In CSRG-111, Computer Systems Research Group (1980), University of Toronto. Also in Proceedings of the Sixth International Conference on Very Large Data Bases, IEEE (1980).

UHLI79 Uhlig, Ronald P., Farber, David J., and Bair, James H., The Office of the Future, North-Holland Publishing Company, Amsterdam, 1979.

APPENDIX

In order to describe ABL succinctly we shall use a modified form of Backus-Naur Form. In the following description, "<Name>" represents a construct, "::=" means "is defined as" and "|" means "or". The construction "(<Name1>, <Name2>)" which appears in the definition of <Step> is used to indicate a selection of one or more of <Name1> and <Name2> without any regard to whether that selection be made in a sequential manner or in a parallel fashion.

```
<System>        ::= <Abstract machine> <Flow> | <Flow> <Abstract machine>

<Abstract machine> ::= <Predicate set> <Abstract machine>
                     | <Action set> <Abstract machine>
                     | <Data set> <Abstract machine> | EMPTY

<Flow>          ::= <Control flow> <Flow> | <Data flow> <Flow> | EMPTY

<Control flow>  ::= <Guard> <Data modification> <Next step> <Control flow> | EMPTY

<Data flow>     ::= <Input data set> <Data modifier> <Output data set> | EMPTY

<Guard>         ::= <Alternative selection> | EMPTY

<Data modification>      ::= <Data modifier> <Data modification> | <Data modifier>

<Next step>     ::= <Step> | <Exception step> | EMPTY

<Alternative selection> ::= <Evaluate> <Step>

<Data modifier>     ::= <Action> | <Alternative> | <Step> | <System>

<Exception step>    ::= <Step>

<Step>          ::= (<Alternative>, <Step>) | EMPTY

<Alternative>   ::= <Action> <Alternative> | EMPTY

<Action set>    ::= <Action> <Action set> | EMPTY

<Evaluate>      ::= Implementation dependent selection
                    according to an evaluation of <Predicate set>.

<Predicate set>     ::= <Predicate> <Predicate set> | EMPTY

<Input data set>    ::= <Data set> | EMPTY

<Output data set>   ::= <Data set> | EMPTY

<Data set>      ::= <Data object> <Data set> | EMPTY

<Predicate>     ::= <Description> | <System>

<Action>        ::= <Description> | <System>

<Data object>   ::= <Description> | <System>

<Description>   ::= <Host language code> <Description>
                  | <Narrative> <Description> | EMPTY

<Host language code>     ::= Set of instructions executable on a hardware machine.

<Narrative>     ::= Set of statements in a natural language.
```

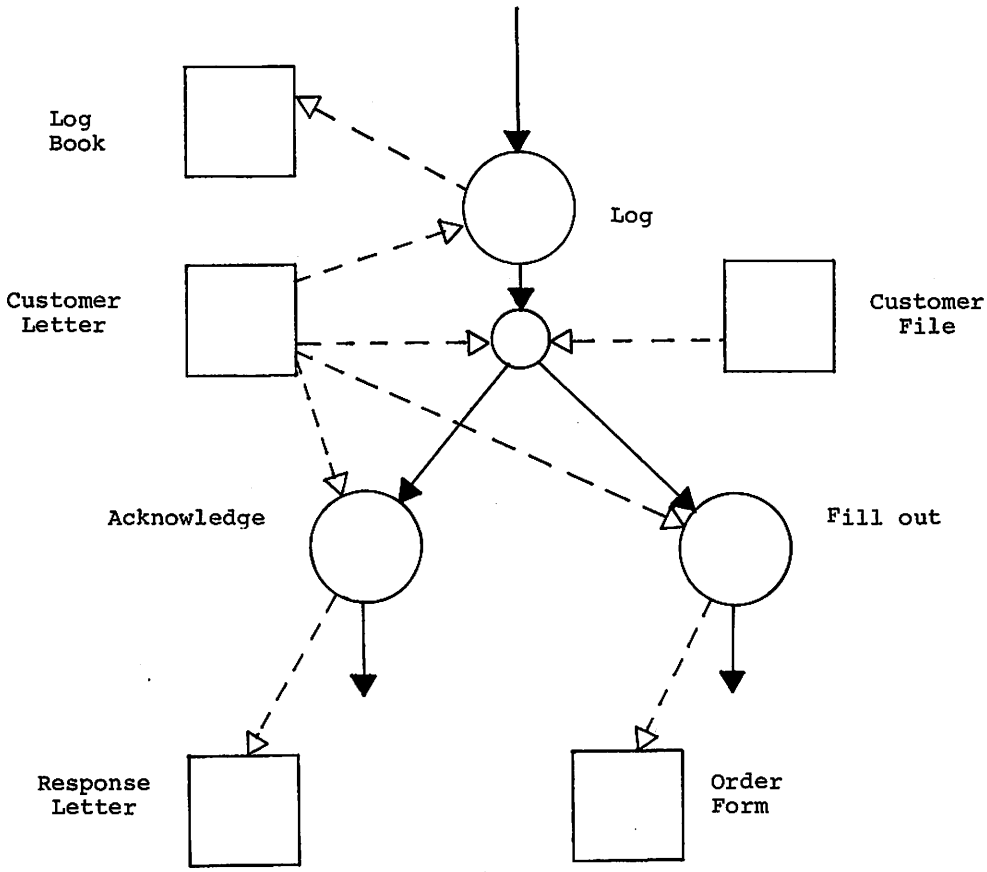


FIGURE 1

AN INFORMATION CONTROL NET (ELLI79)

```
==========================================================================
THE ORDER PROCESSING DEPARTMENT OF A COMPANY
BASED ON A DESCRIPTION IN ELLI79, P.226,P.232

    1. INITIALIZE
<  2, 2, 1 >


    2. SHIPPING INSTRUCTIONS
<  0, 0, 0, 0 >
```

FIGURE 2a

```
==========================================================================
THE ORDER PROCESSING DEPARTMENT OF A COMPANY
BASED ON A DESCRIPTION IN ELLI79, P.226,P.232

    1. INITIALIZE
<  2, 2, 1 >

         1.1 CUSTOMER REQUEST ARRIVES

         1.2 CUSTOMER REQUEST ARRIVES

         1.3 NOT  CUSTOMER REQUEST ARRIVES
    2. SHIPPING INSTRUCTIONS
<  0, 0, 0, 0 >

         2.1 SEND C.O.D.  AND
               NOT  BILL LATER

         2.2 NOT  SEND C.O.D.  AND
               BILL LATER

         2.3 NOT  SEND C.O.D.  AND
               NOT  BILL LATER

         2.4 SEND C.O.D.  AND
               BILL LATER
```

FIGURE 2b

```
==========================================================================
THE ORDER PROCESSING DEPARTMENT OF A COMPANY
BASED ON A DESCRIPTION IN ELLI79, P.226,P.232

    1. INITIALIZE
<  2, 2, 1 >

         1.1 CUSTOMER REQUEST ARRIVES
              1.1 PREPARE ORDER FOR DELIVERY      ----> 2

         1.2 CUSTOMER REQUEST ARRIVES
              1.2 ACKNOWLEDGE RECEIPT OF ORDER      ----> 2

         1.3 NOT  CUSTOMER REQUEST ARRIVES
              1.3 WAIT FOR AN ORDER      ----> 1
    2. SHIPPING INSTRUCTIONS
<  0, 0, 0, 0 >

         2.1 SEND C.O.D.  AND
               NOT  BILL LATER
              2.1 SHIP C.O.D.      ----> 0

         2.2 NOT  SEND C.O.D.  AND
               BILL LATER
              2.2 SHIP NOW, BILL LATER      ----> 0

         2.3 NOT  SEND C.O.D.  AND
               NOT  BILL LATER
              2.3 IMPROPER DELIVERY METHOD      ----> 0

         2.4 SEND C.O.D.  AND
               BILL LATER
              2.4 IMPOSSIBLE DELIVERY METHOD      ----> 0
```

FIGURE 2c

```
==========================================================================
THE ORDER PROCESSING DEPARTMENT OF A COMPANY
BASED ON A DESCRIPTION IN ELLI79, P.226,P.232

    1. INITIALIZE
<  2, 2, 1 >

        1.1 CUSTOMER REQUEST ARRIVES
             1.1 PREPARE ORDER FOR DELIVERY      ----> 2
                  1.1A  LOG CUSTOMER IN LOG BOOK
                  1.1A  TYPE THE ORDER
                  1.1A  SEND THE ORDER
                  1.1A  RECEIVE THE ORDER
                  1.1A  VERIFY ORDER
                  1.1A  PROCESS ORDER


        1.2 CUSTOMER REQUEST ARRIVES
             1.2 ACKNOWLEDGE RECEIPT OF ORDER      ----> 2
                  1.2A  SEND ACKNOWLEDGEMENT LETTER


        1.3 NOT  CUSTOMER REQUEST ARRIVES
             1.3 WAIT FOR AN ORDER      ----> 1

    2. SHIPPING INSTRUCTIONS
<  0, 0, 0, 0 >

        2.1 SEND C.O.D.  AND
             NOT  BILL LATER
             2.1 SHIP C.O.D.      ----> 0
                  2.1A  FILL OUT FORM F3
                  2.1A  SEND GOODS C.O.D.


        2.2 NOT  SEND C.O.D.  AND
             BILL LATER
             2.2 SHIP NOW, BILL LATER      ----> 0
                  2.2A  FILL OUT FORM F1
                  2.2A  FILL OUT FORM F2
                  2.2A  BILL LATER


        2.3 NOT  SEND C.O.D.  AND
             NOT  BILL LATER
             2.3 IMPROPER DELIVERY METHOD      ----> 0
                  2.3A  CALL ERROR HANDLING PROGRAM


        2.4 SEND C.O.D.  AND
             BILL LATER
             2.4 IMPOSSIBLE DELIVERY METHOD      ----> 0
                  2.4A  CALL ERROR HANDLING PROGRAM
```

FIGURE 3

```
THE ORDER PROCESSING DEPARTMENT OF A COMPANY
BASED ON A DESCRIPTION IN ELLI79, P.226,P.232

      1 % COMPUT ( CUSTOMER REQUEST ARRIVES
                 ) OF   --->  <  2, 2, 1 >
           [Y] %  LOG CUSTOMER IN LOG BOOK
                  TYPE THE ORDER
                  SEND THE ORDER
                  RECEIVE THE ORDER
                  VERIFY ORDER
                  PROCESS ORDER
                                                  ---> 2
           [Y] %  SEND ACKNOWLEDGEMENT LETTER
                                                  ---> 2
           [N] %                                  ---> 1

      2 % COMPUT ( SEND C.O.D. ,
                   BILL LATER
                 ) OF   --->  <  0, 0, 0, 0 >
           [Y,N] %  FILL OUT FORM F3
                    SEND GOODS C.O.D.
                                                  ---> 0
           [N,Y] %  FILL OUT FORM F1
                    FILL OUT FORM F2
                    BILL LATER
                                                  ---> 0
           [N,N] %  CALL ERROR HANDLING PROGRAM
                                                  ---> 0
           [Y,Y] %  CALL ERROR HANDLING PROGRAM
                                                  ---> 0
```

FIGURE 4

THE ORDER PROCESSING DEPARTMENT OF A COMPANY
BASED ON A DESCRIPTION IN ELLI79, P.226,P.232

- STEP DESCRIPTION -

1.0 INITIALIZE

1.1 PREPARE ORDER FOR DELIVERY	2
1.2 ACKNOWLEDGE RECEIPT OF ORDER	2
1.3 WAIT FOR AN ORDER	1

2.0 SHIPPING INSTRUCTIONS

2.1 SHIP C.O.D.	0
2.2 SHIP NOW, BILL LATER	0
2.3 IMPROPER DELIVERY METHOD	0
2.4 IMPOSSIBLE DELIVERY METHOD	0

CONTROL FLOW									MACHINE	
C1	V	V	V	.	.	.	.	V		
C2	.	.	.	V	V	V	V	V		
P1	Y	Y	N	-	-	-	-	-	P1	CUSTOMER REQUEST ARRIVES
P2	-	-	-	Y	N	N	Y	-	P2	SEND C.O.D.
P3	-	-	-	N	Y	N	Y	-	P3	BILL LATER
A1	1	.	.	.	.	.	.	.	A1	LOG CUSTOMER IN LOG BOOK
A12	.	1	.	.	.	.	.	.	A12	SEND ACKNOWLEDGEMENT LETTER
A9	.	.	.	1	.	.	.	.	A9	FILL OUT FORM F3
A7	.	.	.	.	1	.	.	.	A7	FILL OUT FORM F1
A13	.	.	.	.	.	1	1	.	A13	CALL ERROR HANDLING PROGRAM
A2	2	.	.	.	.	.	.	.	A2	TYPE THE ORDER
A5	.	.	.	2	.	.	.	.	A5	SEND GOODS C.O.D.
A8	.	.	.	.	2	.	.	.	A8	FILL OUT FORM F2
A3	3	.	.	.	.	.	.	.	A3	SEND THE ORDER
A6	.	.	.	.	3	.	.	.	A6	BILL LATER
A4	4	.	.	.	.	.	.	.	A4	RECEIVE THE ORDER
A10	5	.	.	.	.	.	.	.	A10	VERIFY ORDER
A11	6	.	.	.	.	.	.	.	A11	PROCESS ORDER
NEXT	2	2	1	0	0	0	0	0		

FIGURE 5

DEVELOPMENT OF A CBMS MESSAGE TRANSFER PROTOCOL

Gerard F. Mulvenna
National Bureau of Standards
Building 225, Room B226
Washington, DC 20234

In recent years, organizations have been turning towards the use of automation equipment to effect productivity improvements in the office environment. One of the principal new methods of expediting communications in the modern office is the use of electronic mail and messaging systems. According to International Resource Development, Incorporated, the market for electronic mail and message systems and services will increase from about $1 billion in 1979 to more than $4.5 billion by 1989. By the latter date, there will be at least 50 companies with significant participation in electronic mail. The declining services and rising costs associated with postal deliveries along with the benefits of "instant" message delivery using electronic means, have sparked a large increase in the use of electronic mail and messaging systems by United States corporations. The growth in electronic mail can be explained by the belief that todays' businesses have to communicate far more effectively in order to compete in an increasingly complex world.

One of the principal types of messaging systems being developed for the office of the future is the computer based message system (CBMS). A CBMS provides three general functions: message creation, message transfer and recipient processing. A CBMS provides software which assists the message originator in creating a message at his terminal. The CBMS offers text editing facilities which allows the originator to review and modify the message before sending it. When the message is ready to be posted, the message originator passes the message content together with a list of the message recipients to that part of the CBMS which transfers the message. When the message arrives it is stored in a message queue until the recipient logs on to his terminal and requests his mail. The recipient is then informed that mail has arrived, and is given a short summary of the subject matter of each message. The CBMS also provides software which allows the message recipient to read, print, store and retrieve messages.

A CBMS can be distinguished from other types of messaging systems. A CBMS provides more than terminal device to terminal device communications. In a CBMS the originators and recipients of messages are terminal users or processes. Another distinguishing characteristic of a CBMS is that it is a store and forward system. The computer system that serves the message recipient need not be available when the originator sends a message; the message can be stored in the originator's system or at an intermediate node in the network until the recipient's system becomes available.

As the number of CBMSs marketed by different vendors increases, it is only natural that the users of different CBMSs will want to communicate with each other. The Institute for Computer Sciences and Technology (ICST) of the National Bureau of Standards, as part of its program [A], of developing network protocol standards is developing a message transfer protocol to control the flow of information between heterogeneous CBMSs and to provide specialized services [B]. A message transfer protocol governs how a message is moved from the originator's CBMS to the recipient's CBMS. It does not specify how messages are created nor what happens to the messages after transfer is completed. The CBMS message transfer protocol includes a specification of the structure and content of the message envelope. It also specifies how a message originator identifies a message recipient and how a CBMS uses that information to correctly deliver the message.

This paper presents the key issues involved in developing a CBMS message

transfer protocol and discusses the work being done in this area by the ICST and by other standards organizations through the use of a logicl model of a CBMS, which is useful for providing a framework for the development of a message transfer protocol.

CBMS Model

The Institute for Computer Sciences and Technology has adopted the CBMS model originally developed by the International Federation for Information Processing Working Group 6.5 (IFIP). The model is illustrated in Fig 1. The User Agent is a set of computer processes which the user accesses to create, present, and store messages. The Message Transfer System is an entity that accepts a message from the originator's User Agent and ultimately passes it to each recipient's User Agent. The originator's User Agent submits the message to the Message Transfer System through the "posting slot". The "posting slot" is the point at which responsibility for a message passes from the originator's User Agent to a Message Transfer System which is composed of one or more cooperating functional components called Message Transfer Agents. The originator's Message Transfer Agent passes the message to the recipient's Message Transfer Agent or to some intermediate Message Transfer Agent which is closer to the destination. The message ultimately arrives at the recipient's Message Transfer Agent and is delivered through the "delivery slot" to the recipient's User Agent. The delivery slot is the point at which responsibility for the message passes from the Message Transfer System to the recipient's User Agent. The recipient's User Agent then makes the message available to the recipient. The User Agent and Message Transfer Agent can be located in the same processor or they can be located in different processors supplied by different vendors.

Using this model, a message can be divided into two parts: the message content and the message envelope. The message content is the information that the originator wishes to send to the recipient. The message envelope consists of all the information required by the Message Transfer System.

Naming and Addressing

Because the recipients of messages in a CBMS are normally people, it is important to develop a specification for naming and addressing these recipients as part of a CBMS Message Transfer Protocol. A name is a collection of attributes which serves to identify uniquely a message recipient within a given naming domain. The attribute types and values are structured so as to be remembered easily by the users of the message system. Each naming domain has a naming authority which has the sole responsibility for issuing names and guaranteeing that the names are unique within its jurisdiction. The concatenation of a name assigned by a naming authority and the name of the naming domain produces a globally unique name. In some cases, the personal name attribute is sufficient for the naming authority to guarantee uniqueness. In other cases, certain organizational and geographical attributes may also be required, (e.g., the John Jones who works for Company A or the Mary Smith who resides on Elm Street). The name may refer to a distribution list rather than to a single recipient. A distribution list might represent a set of users with a particular interest, or users who belong to a particular organization.

Whether the name identifies a single recipient or a distribution list, the name must be unique. The property of uniqueness is important because in order to deliver a message, the name must be mapped into an address which identifies a specific delivery slot in the message system. The address is structured so that it reflects the structure of the Message Transfer System and can be easily interpreted by a computer. The address is not necessarily structured so that it is easily remembered by a user of the message system.

Since the information that the originator knows about a recipient and the information that a CBMS needs to deliver the message are usually different, a CBMS must provide a directory service to transform attributes and names into addresses. The name to address mapping can take place before the message is posted or the address can be resolved while the message is en route to the destination.

Interconnected systems that serve a small and relatively stationary user group may choose to perform address resolution before the message is posted. The originator may query a local or remote directory to obtain the address information. If the address information is stored in a remote directory, the query to obtain the information might be structured in the form of a message sent to a remote process. Once the address information is obtained, it can be saved

and used by the originator in subsequent correspondence to the same recipient.

The method of posting the message with name information only and resolving the address en route to the destination also has advantages. The knowledge of address information is more centralized and consequently there are fewer directory update problems. This is particularly important when the user group is mobile. An originator who has a high level of confidence in the name information that he is supplying may want to use this alternative.

The method of resolving the address before the message is posted and the method of resolving the address en route to the destination are not mutually exclusive. Both methods can be available for use by the originator as circumstances dictate.

The number and placement of the directories that serve interconnected systems may also vary. Possible implementation alternatives include a centralized directory, a directory for each naming domain and a directory residing in each Message Transfer Agent.

Services Provided to CBMS Users

In the process of transferring a message from an originator to a recipient, a CBMS provides certain service options to its users. These services determine the content and structure of the message envelope as well as the design of the message protocols. Following is a list of services under consideration by ICST and a brief discussion of each service; not all services will be provided in every CBMS.

Multi-Destination Delivery: allows the originator's User Agent to post a message to more than one recipient.

Timed Delivery: provides the originator's User Agent with the capability of constraining the date and time of message delivery.

Priority Delivery: provides for expedited transfer and delivery of messages.

Posting Time Stamp: provides the User Agents with the date and time at which a message was submitted by the originator's User Agent to the Message Transfer System.

Delivery Time Stamp: provides the User Agents with the date and time at which a message was delivered by the Message Transfer System to the recipient's User Agent.

Message Identification: a service by which the Message Transfer System provides a unique identification for each message on the message envelope. The message identification can be used by the User Agents to retrieve messages and by the Message Transfer Systems to provide delivery and nondelivery notification.

Originator's User Agent Content Type Indication: provides the originator's User Agent with the capability of specifying the content type of the posted message to the Message Transfer System.

Recipient's User Agent Content Type Indication: provides the recipient's User Agent with the capability of specifying the content types for which it will accept delivery.

Content Type Conversion: allows the User Agents to specify that the Message Transfer System perform a conversion of the message content before delivery. The requested conversion may be from one format to another (e.g., CBMS to Teletex) or from one medium to another (e.g., voice to text).

Nondelivery Notification: provides the originator's User Agent with a notification that the message could not be delivered to the recipient's User Agent. The reasons for nondelivery include a recipient address that is unknown to the Message Transfer System, a message content type that is unacceptable to the recipient's User Agent and a time delivery message that cannot be delivered within the specified time constraints.

Delivery Notification: provides the originator's User Agent with notification that the message has been successfully delivered to the recipient's User Agent.

Receipt Notification: allows the originator of a message to request notification of the receipt of the message by its recipient.

Redirection of the Message by the Recipient's User Agent: allows the recipient's User Agent to specify an alternate recipient User Agent to whom the message should be

delivered. The recipient may want another person to receive his messages in his absence or he may want messages temporarily rerouted to him at another location.

Address Information Inquiry: provides the User Agent with the address associated with the name of a potential message recipient. The address is determined by consulting a database which associates names with addresses. An address information inquiry is initiated by a User Agent and processed by the Message Transfer System. There are a number of possible responses to an address information inquiry. If there is no potential recipient associated with the name, a failure response is returned. If the name can be uniquely associated with a potential recipient, the recipient's address is returned. If the input parameters (e.g., last name = Jones), can be associated with more than one potential recipient, the requestor can ask either for a failure indication or a list of all names and addresses that match the input parameters.

Inquiry Processing: provides information of concern to the user (e.g., billing rates) upon request. The Message Transfer System will unilaterally notify the User Agents when an abnormal condition occurs in the system.

The message transfer protocol will specify which of these services must be implemented by every interconnected CBMS. However, the protocol must also be flexible enough to allow CBMS's to provide services which are not categorized as essential.

In order to a transfer a message from an originator to a recipient and to provide services requested by the user, interaction is required between the User Agent and the Message Transfer System. In order to describe how this interaction might take place and to identify the protocols required by a CBMS to perform its task, a layered version of the IFIP CBMS model was developed by the International Telegraph and Telephone Consultative Committee (CCITT) Message Handling Systems group.

Layered Model of CBMS

In the International Organization for Standardization (ISO) Reference Model for Open Systems Architecture [C], a layer provides services to users in the next higher layer. In order to provide its services, a layer builds its functions on the services it requires from the next lower layer. The interactions between a given layer and the layer below which provides its services are called service primitives. The points of interaction between layers are called service access points. Issuing a service primitive at a service access point does not in itself assure that the requested service is performed. If the initiating entity needs to rely on the completion of the requested action, it must wait for a confirmation to be received from the requested entity.

In the framework of the ISO Reference Model, User Agents (UA) and Message Transfer Agents are application layer entities. In the layered CBMS model, illustrated in figure 2, the application layer is further divided into two sublayers, the User Agent Sublayer and the Message Transfer Sublayer. The User Agent Sublayer is the upper sublayer. It is concerned with the processing that User Agents perform on the message contents at the request of the user. The originator and recipient User Agents are the peer entities communicating at this sublayer. The Message Transfer Sublayer is the lower sublayer and is concerned with the work that must be performed to move a message from an originator's User Agent to a recipient's User Agent.

The Message Transfer Sublayer does not provide the reliable transfer of a message from one MTA to another. It relies upon lower layer service providers to transmit and acknowledge data units, perform error detection and flow control, and perhaps perform certain data transformations as the message is transmitted between MTAs. Since the address supplied by the User Agent Sublayer does not specify the route that the message should take through the Message Transfer System, the Message Transfer Sublayer maps this address into the address of the MTA entity to which the message should be routed. This is repeated at each intermediate MTA until the message arrives at the destination MTA.

Three types of systems can be represented using this layered logical CBMS model: systems that contain User Agents only, systems that contain Message Transfer Agents only and systems that contain both User Agents and Message Transfer Agents.

Three functional entities appear in the layered model: the User Agent entity, the Message Transfer Agent entity, and the Message Transfer System Access Entity (MTSAE). The MTSAE is the Message Transfer Sublayer software which resides in the same system as the User Agent and allows the User Agent to have access to the services of the Message Transfer System. The MTSAE is the User Agent communications software; it provides a User Agent with access to the services of a Message Transfer Agent which may be colocated in the same system or in a different system.

Three protocols appear in the layered model. Protocol P1 defines the interaction between Message Transfer Agents to relay messages and to provide Message Transfer Sublayer services. The message envelope contains all the information required by the Message Transfer Agents to do this.

Protocol P2 defines the interaction between cooperating User Agents to provide the User Agent Sublayer services required by the CBMS user. An example of a cooperating User Agent service is a receipt notification which could be generated by the recipient's User Agent and transmitted to the originator's User Agent when the recipient first examines the message.

Protocol P3 defines the interaction between the originator's MTSAE and the posting Message Transfer Agent or between the delivering Message Transfer Agent and the recipient's MTSAE. The P3 protocol is not required if the User Agent and Message Transfer Agent are in the same system.

The Message Transfer System interfaces with many types of User Agents. The User Agent communications software (MTSAE) is likely to be provided by a different vendor than the MTA vendor. The MTA may not have the same degree of confidence in information received from an MTSAE that it would have if the information were received from another MTA. More integrity assurance may be required and the protocol may be different, (i.e., P3 protocol instead of P1 protocol). The protocol between an MTSAE and a Message Transfer Agent which is not in the same system may be the same as the MTA-MTA protocol; the conditions which cause it to be different have not yet been determined.

The service access points in the layered CBMS model are the posting and delivery slots referred to earlier. The service access points can also be used to pass information between two sublayers. For example, the User Agent Sublayer can query the Message Transfer Sublayer for billing information and the Message Transfer Sublayer can unilaterally provide information about the status of the Message Transfer System or the status of a particular message.

A list of the service primitive groups that describe the interaction between the Message Transfer Sublayer (MTSL) and the Message Transfer Sublayer user (i.e., the User Agent Sublayer) might include the following:

SUBMIT - primitives by which an originating MTSL user initiates the transfer of a message to one or more destination MTSL users.

DELIVER - primitives by which the MTSL passes a previously submitted message to the destination MTSL user.

QUERY - primitives by which an MTSL user obtains information upon request from the MTSL, e.g., tariff information.

INFORM - primitives by which the MTSL user informs the MTSL about some aspect of the MTSL user's status, e.g., request to redirect incoming messages to an alternate recipient User Agent.

STATUS - primitives by which the MTSL informs the MTSL user about an event or condition, e.g., nondelivery notification.

Space constraints will not permit an extensive discussion of all these service primitive groups. However, in order to understand the interaction between the MTSL and the MTSL user when a message is posted, let us examine the SUBMIT service primitive in detail. There are two events associated with the SUBMIT service primitive: SUBMIT-REQUEST and SUBMIT-CONFIRM.

A SUBMIT-REQUEST is a call issued by the MTSL user to the MTSL requesting that a message be posted. Its parameters include the following:

DESTINATION USER AGENT ADDRESS(ES)
MESSAGE CONTENT
LIST OF REQUESTED SERVICES

If the name to address resolution is performed while the message is en route to the destination, the Destination User Agent Name can be

substituted for the Destination User Agent Address.

The MTSL user will specify the services that it is requesting from the Message Transfer Sublayer when posting a message. Options include a normal delivery, priority delivery, timed delivery, multi-destination delivery, content type conversion and delivery or nondelivery notification. The MTSL may provide certain of these services by default (e.g., normal delivery, nondelivery notification) but require that other services (e.g., priority delivery) be specified as parameters in the SUBMIT REQUEST call.

A SUBMIT-CONFIRM is issued by the MTSL in response to a SUBMIT- REQUEST. Its parameters include the following:

SUCCESS INDICATION

and, if SUCCESS INDICATION is positive,

SUBMIT-TIME STAMP
MESSAGE ID.

SUBMIT-TIME STAMP indicates when the MTSL accepted the SUBMIT-REQUEST and that the message was posted. The Message ID is assigned by the MTSL and returned to the MTSL user with the posting confirmation. The Message ID will be used by the MTSL in subsequent correspondence with the MTSL user to reference a message (e.g., to identify the message that could not be delivered in a nondelivery notification.

If the SUCCESS-INDICATION is negative, there will be a FAILURE REASON parameter to indicate why SUBMIT-REQUEST was rejected by the MTSL.

Summary

This paper has defined a computer based message system and presented a brief overview of the work being done by the ICST and other standards organizations that are developing a message transfer protocol to allow interconnection of heterogeneous CBMSs.

The CBMS message transfer protocol is still in the early stages of development. Much work remains to be done in specifying the structure and content of the message envelope, the form of a name and address and the methods of communicating with a directory service that will perform the name to address transformation. The Institute for Computer Sciences and Technology will continue to work closely with other national and international standards organizations to develop a protocol which will serve the needs of the users and service providers.

REFERENCES

[A] Blanc, R. P. and Heafner J. F. "Off the shelf solutions motivate NBS Standards Drive," Data Communications, pg 223-229, March 1982.

[B] Features of a Message Transfer Protocol, National Bureau of Standards, Proposed Federal Information Processing Standard, September 1981.

[C] ISO/TC97/SC16 Reference Model of Open System Interconnection Draft Proposed ISO/DIS 7498, International Organization for Standardization, December 1980.

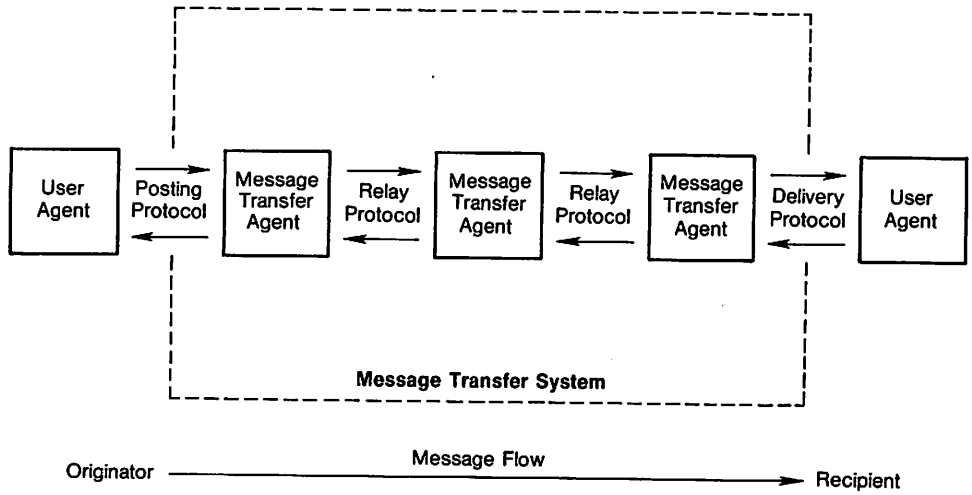


Figure 1 Logical Model of A Computer Based Message System

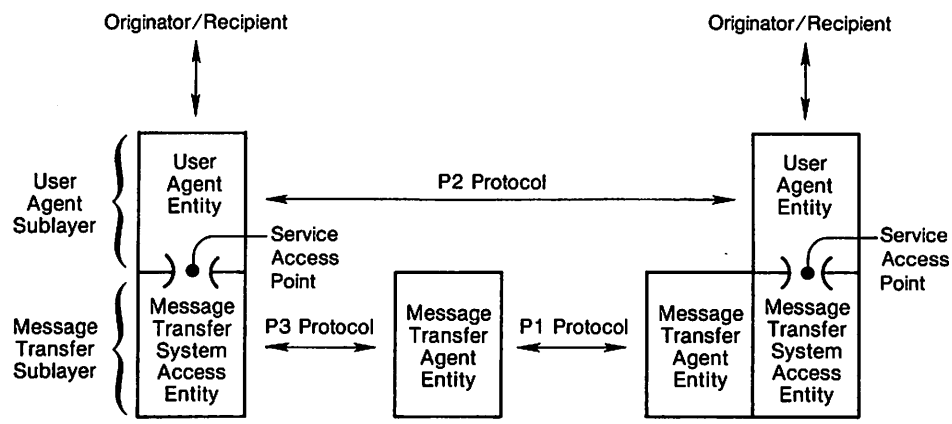


Figure 2 Layered Model of A Computer Based Message System

SESSION 10

PANEL: HOW MUCH BANDWIDTH IS ENOUGH?

Chairperson: Karle Kümmele
IBM, Zurich

Communications may be thought of as the glue that coordinates activities within and between offices. As tasks within the office have become more computerized, the nature of communications has changed, moving from primarily face-to-face voice communication in the "primitive" office to a mixture of voice and data today. The mixture could change again as graphics, and perhaps video, become more commonly used.

Office communication requirements depend very much on the specific office scenario that one assumes. If one envisages communicating word processors with the existing pattern of voice communication then perhaps a 3 Mbit/sec local network working in conjunction with a voice PBX would be adequate. If, on the other hand, integrated voice, data, facsimile and video services are envisaged, then even channel capacities in excess of 100 Mbit/sec might be inadequate.

The panel will consider both the impact of various office scenarios on communications needs and the technology which can be used to meet these needs.

AUTHOR INDEX

N. Archer, Digital Equipment Corporation 29
G. Attardi, Istituto di Scienze dell'Informazione 52
L. Baer, Xerox Corporation 1
A. D. Bailey, Jr., University of Minnesota 27
G. R. Barber, MIT 79
J. Barron, University of Toronto 12
J. L. Bennett, IBM 101
M. Bernal, Xerox PARC 131
C. V. Bullen, MIT 101
R. J. Byrd, IBM 67
E. D. Carlson, IBM 101
R. Choudhari, State University of New York 108
S. Christodoulakis, University of Toronto 110
G. Curry, Xerox Corporation 1
S. P. deJong, MIT 67,122
C. A. Ellis, Xerox PARC 131
J. Farrell, Xerox Corporation 10
J. C. Ferrans, Western Electric Company 123
J. Gerlach, Purdue University 27
S. J. Gibbs, University of Toronto 21
M. Good, Digital Equipment Corporation 29
W. Harless, Bank of America, Integrated Office Systems Division 50
J. J. Hull, State University of New York 108
W. M. Jaworski, Concordia University 141
P. Klose, Xerox Corporation 10
K. Kümmele, IBM, Zurich 160
J. Lebensold, Concordia University 141
B. Lee, Xerox Corporation 1
D. L. Lee, University of Toronto 113
D. Lipkie, Xerox Corporation 1
T. W. Malone, Xerox PARC 47
N. F. Maxemchuk, Bell Laboratories 41
R. P. McAfee, University of Western Ontario 27
G. F. Mulvenna, National Bureau of Standards 153
N. Naffah, INRIA 11
R. R. Panko, University of Hawaii 82
R. Purvy, Xerox Corporation 10
T. Radhakrishnan, Concordia University 141
M. Simi, Istituto di Scienze dell'Informazione 52
S. E. Smith, IBM 67
R. H. Sprague, Jr., University of Hawaii 82
S. N. Srihari, State University of New York 108
D. Tsichritzis, University of Toronto 110
J. H. Tucker, Georgia State University 93
A. B. Whinston, Purdue University 27
J. Whiteside, Digital Equipment Corporation 29
H. A. Wilder, Bell Laboratories 41
D. Wixon, Digital Equipment Corporation 29

APPLICATION FOR SIG MEMBERSHIP*

ASSOCIATION FOR COMPUTING MACHINERY
11 WEST 42nd STREET, NEW YORK, N.Y. 10036

Please enroll me as a member of the SPECIAL INTEREST GROUP ON OFFICE AUTOMATION

Membership includes Newsletter subscription. Please make checks payable to: ACM, Inc., P.O. Box 12115, Church Street Station, New York, N.Y. 10249

Name (Please print or type)

Mailing Address

City State Zip

Signature

☐ New Address. Please change my ACM record.

***NOTE:** For ACM members renewing within the next three months, do not use this application. Simply add this SIG to your renewal invoice when you receive it and return with the appropriate additional payment.

Annual Membership Dues are $7.50 for ACM Members, $3.00 for ACM Student Members and $21.50 for Non-ACM Members.

☐ **ACM MEMBER**
ACM Member No. _____
Send no money now.
Dues are payable when ACM membership is renewed.

☐ **ACM STUDENT MEMBER**
ACM Student Member No. _____
Send no money now.
Dues are payable when ACM membership is renewed.

☐ **NON-ACM MEMBER**
Enclosed is annual dues of $21.50

☐ **SUBSCRIPTION TO SIG OA**
NEWSLETTER ONLY
Enclosed is payment for annual subscription of $17.50
Note: Subscription is included with membership dues.

☐ Please send information on ACM Membership.

Association for Computing Machinery
11 WEST 42nd STREET, NEW YORK, NY 10036